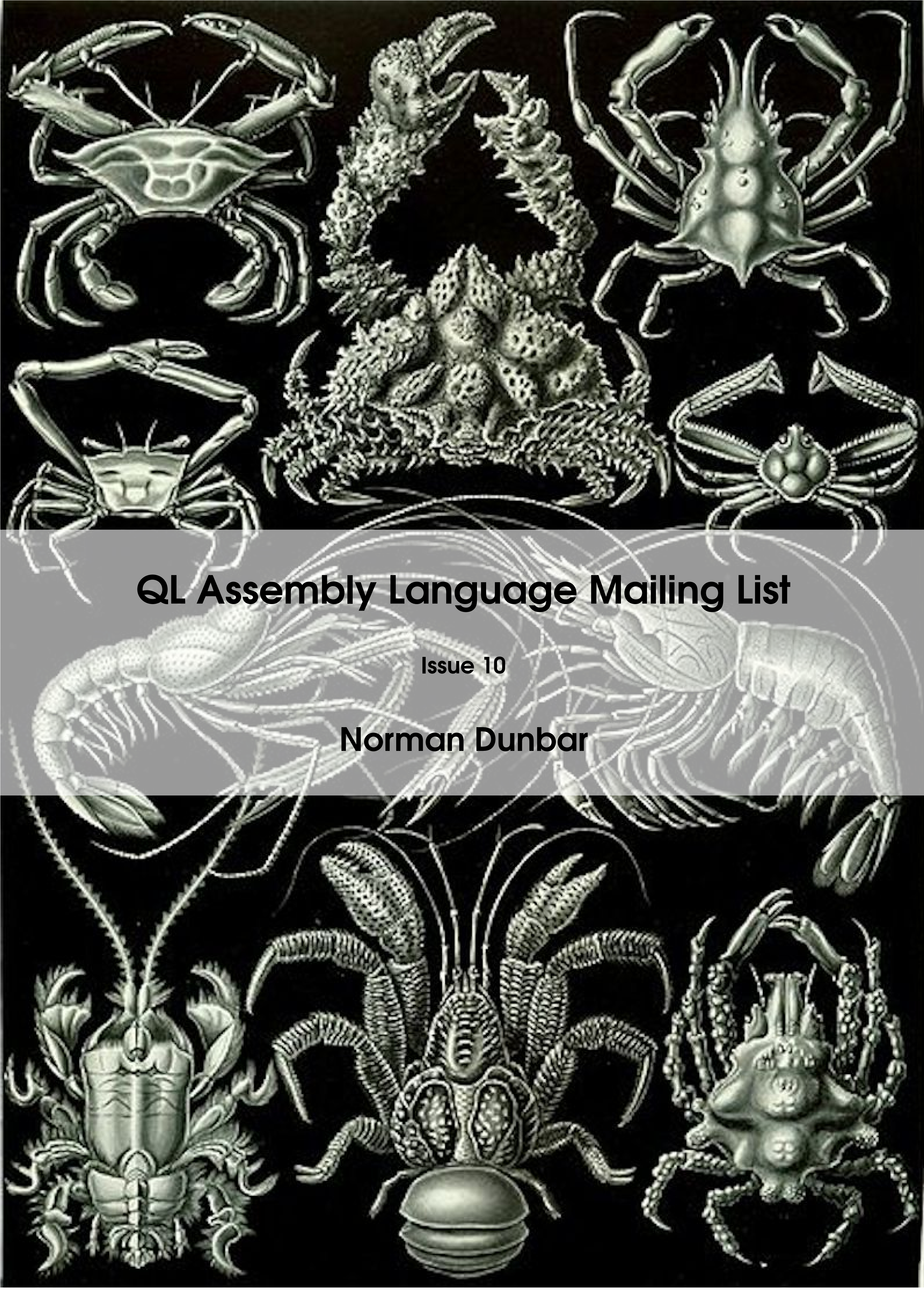


QL Assembly Language Mailing List

Issue 10

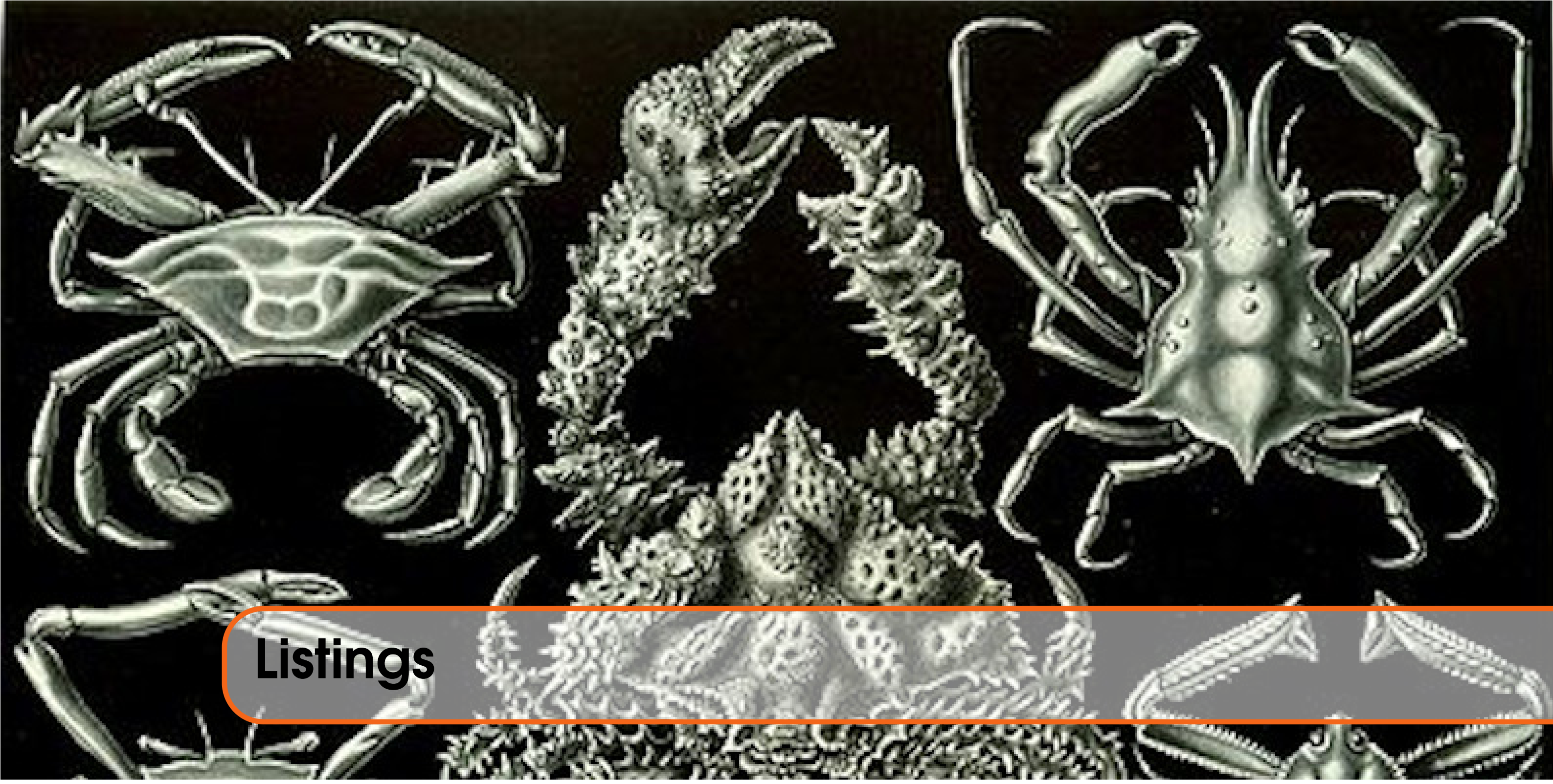
Norman Dunbar

PUBLISHED BY MEMYSELFEYE PUBLISHING ;-)

Download from:
https://github.com/NormanDunbar/QLAssemblyLanguageMagazine/releases/tag/Issue_10

This pdf document was created on *29/1/2022* at *16:57:02*.

Contents



Listings

1. Preface

1.1 Feedback

Please send all feedback to assembly@qdosmsq.dunbar-it.co.uk. You may also send articles to this address, however, please note that anything sent to this email address may be used in a future issue of the eMagazine. Please mark your email clearly if you do not wish this to happen.

This eMagazine is created in LaTeXsource format, aka plain text with a few formatting commands thrown in for good measure, so I can cope with almost any format you might want to send me. As long as I can get plain text out of it, I can convert it to a suitable source format with reasonable ease.

I use a Linux system to generate this eMagazine so I can read most, if not all, Word or MS Office documents, Quill, Plain text, email etc formats. Text87 might be a problem though!

1.2 Subscribing to The Mailing List

This eMagazine is available by subscribing to the mailing list. You do this by sending your favourite browser to http://qdosmsq.dunbar-it.co.uk/mailinglist and clicking on the link “Subscribe to our Newsletters”.

On the next screen, you are invited to enter your email address *twice*, and your name. If you wish to receive emails from the mailing list in HTML format then tick the box that offers you that option. Click the Subscribe button.

An email will be sent to you with a link that you must click on to confirm your subscription. Once done, that is all you need to do. The rest is up to me!

1.3 Contacting The Mailing List

I'm rather hoping that this mailing list will not be a one-way affair, like QL Today appeared to be. I'm very open to suggestions, opinions, articles etc from my readers, otherwise how do I know what I'm doing is right or wrong?

I suspect George will continue to keep me correct on matters where I get stuff completely wrong, as before, and I know George did ask if the list would be contactable, so I've set up an email address for the list, so that you can make comments etc as you wish. The email address is:

assembly@qdosmsq.dunbar-it.co.uk

Any emails sent there will eventually find me. Please note, anything sent to that email address will be considered for publication, so I would appreciate your name at the very least if you intend to send something. If you do not wish your email to be considered for publication, please mark it clearly as such, thanks. I look forward to hearing from you all, from time to time.

If you do have an article to contribute, I'll happily accept it in almost any format - email, text, Word, Libre/Open Office odt, Quill, PC Quill, etc etc. Ideally, a LaTeXsource document is the best format, because I can simply include those directly, but I doubt I'll be getting many of those! But not to worry, if you have something, I'll hopefully manage to include it.

2. News

2.1 New Cover

I had a note on the QL Forum that some people[1] get a bit weird in the head when they look at the cover image for this eMagazine. It's got something to do with all the holes, or the appearance of same, so rather than losing any of my valued readers, I've decided to get a new cover image with far fewer holes!

It may sound amusing, but Trypophobia[2] is a known phobia, so holes are best avoided if you are a sufferer. Hopefully the new cover will prevent any induced weirdness in my reader(s). There's enough weirdness in the author!

2.2 My Assembly Book

In the last issue, I mentioned that *Tinyfpga* had created a printed version of my Assembly Language articles from the late, and much missed, *QL Today* magazine. Since then, Tiny (if I can call him that!) has posted a couple of updates on the book to the QL Forum at this post[3] and this post too[4].

It appears that everyone who purchases a copy gets to see the personal details of the previous customer, which means potentially falling foul of EU GDPR rules and regulations for personal data. There is a fix to the problem, but if anyone has a better idea, please do post the details on the forum in response to the two posts linked above. Thanks.

[1] Ok, one person!

[2] https://en.wikipedia.org/wiki/Trypophobia

[3] https://qlforum.co.uk/viewtopic.php?f=3&t=3976&start=10#p44445

[4] https://qlforum.co.uk/viewtopic.php?f=3&t=3976&start=10#p44446

2.3 Beginner's Corner

This issue sees the start of a new feature, which will take a look at Assembly Language programming from a beginner's point of view. I don't mean getting right back to basics and learning the instructions etc – that's nicely covered in the book mentioned above, or the PDF version which you can download from my GitHub repository[5].

What I do mean is a beginner's guide to Assembly Language Tools – which assembler to use, what about debuggers etc, plus, it occurs to me that I never delved into QDOS and the various utilities and traps in my articles for *QL Today*! So that's where we will be going in the future.

2.4 SMSQ/E

And speaking of delving into QDOS, this issue marks the end of an era. QDOS is no more, long live SMSQ/E! From now on, I will be converting myself over to using the SMSQ/E versions of trap calls and vectors etc. I feel, after some discussion on the forum, that anyone learning this stuff nowadays – and there are still some – should be using the up to date details. So, no more using `UT_CON` as I'll be using `OPW_CON` instead!

For those who need a new manual to cover this new regime, the *QDOS/SMS Reference Manual* is available from the Sinclair QL Home Page[6] where you can grab a PDF version[7] or a Libre Office version[8] (ODT) as desired.

2.4.1 SMSQ/E 3.38

A new version of SMSQ/E is now available. Version 3.38 was announced on 1st November on the QL Forums[9] and on the QL emailing list. To quote Wolfgang Lenerz:

SMSQE 3.38 is out now.

As usual, you can get it at http://www.wlenerz.com/smsqe/.

The main news here is that, thanks to Alain Haoui's work, WMAN can now draw real subwindow indices.

The ways this is done is explained in the QPTR manual, the new version of which can be found at my QL stuff site http://www.wlenerz.com/qlstuff.

There you can find the QPTR manual (in the documentation section), and also the new QPTR bin file itself, which also implements index drawing (in the programming section) as well as two demo/test programs (one for Basic, made by Alain. Haoui, and one for assembler).

2.5 QPC2 Version 5.01

Coincidentally with the new release of SMSQ/E, Marcel Kilgus announced the release of QPC2 v5.01 on 1st November 2021. In his own words[10]:

[5] https://github.com/NormanDunbar/QLAssemblyLanguageBook/releases/latest

[6] http://www.dilwyn.me.uk/docs/manuals/index.html

[7] http://www.dilwyn.me.uk/docs/manuals/QDOS_SMS%20Reference%20Guide%20v4.5.pdf

[8] http://www.dilwyn.me.uk/docs/manuals/QDOS_SMS%20Reference%20Guide%20v4.5.odt

[9] https://qlforum.co.uk/viewtopic.php?f=3&t=3957&p=44189#p44189

[10] https://qlforum.co.uk/viewtopic.php?f=19&t=3958#p44190

Rejoice, QPC2 got a new release today. As always free as in "free beer". Details are here: https://www.kilgus.net/2021/11/01/qpc2-v5-01-and-smsq-e-v3-38-released/

And, yes, I know. The chances of a newer release of QPC and/or SMSQ/E appearing *before* this issue of the eMagazine are pretty high!

2.6 Code Listings

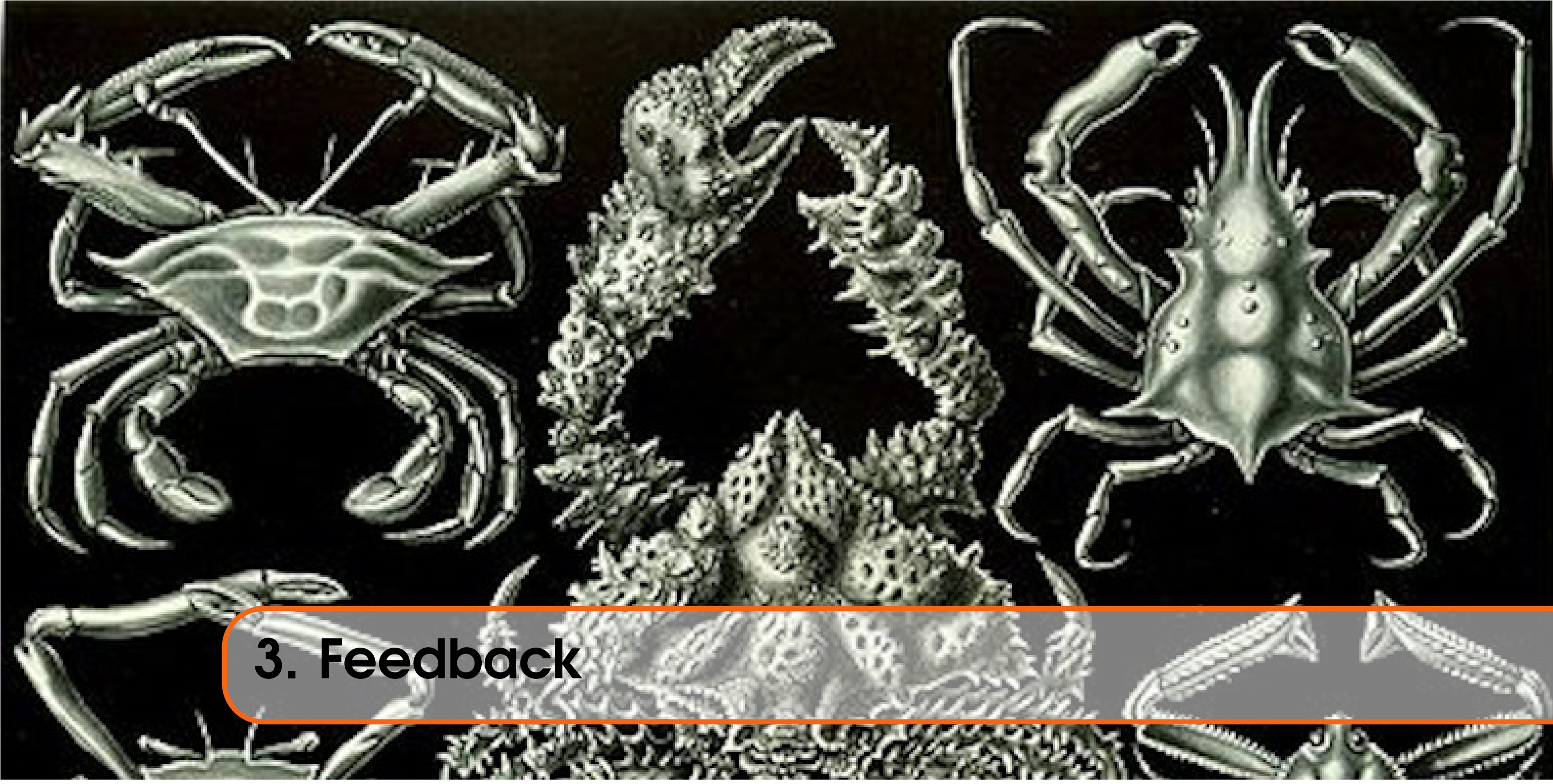
I'm using a different manner of including listings in the eMagazine. From this issue onwards, the vast majority of code will be taken directly from the source files and not copied and pasted as I have been doing. This way, I *should* reduce the number of errors I make when copying and pasting – or forgetting to! Basically, from here on in, the errors anyone finds will be all my own fault!

I'm also not including the bigger comments in the source code, but the source files will still have them present. There's no point having the text explaining things that the source comments are also explaining.

Not all this issue's listings have been converted over.

2.7 And Finally...

Apologies for the seemingly never ending list of URLs in the footnotes to the pages in this section. This is mainly done for the reason that some people actually prefer a printed version of my eMagazine and it appears that there's a bug in the handling of clickable links on the paper versions – they simply don't work.



3. Feedback

3.1 Circular Buffers

When I announce Issue 9 on the QL Forum, Marcel queried the Circular Buffer article in the issue. He was wondering if I had simply recreated the Queue handling features of QDOSMSQ, those being `IOQ.SETQ`, `IOQ.TEST`, `IOQ.PBYT`, and `IOQ.GBYT`.

Putting it simply, *probably*! However, as I pointed out, it was a fun exercise in creating the code from a C++ version, and debugging a particularly insidious bug where allocating common heap space gets rounded up!

Thanks Marcel, at least I know one person read it!

3.2 Learning Assembly Language

TMD2003 wondered about learning Assembly Language and started a thread[1] on the QL Forum[2]. On the second page of the thread, *Tinyfpga* issued this statement:

In Basic all I have to do start my journey into the programming world is to:- Type into QD

```
OPEN #1,con_
OUTLN #1,310,60,50,300
CLS #1 : BORDER #1,1,7
INPUT #1,a$
```

I then save the text in a ram disk, press the execute button and ,"hey presto", I see an app on my screen. What could be an easier introduction to programming?

[1] https://qlforum.co.uk/viewtopic.php?f=3&t=3976

[2] Yes, I know, it's not really feedback on the previous issue of the eMagazine, but I thought it was relevant.

Well, I decided that that would be a good start to the new Beginner's feature, so I've taken up *Tinyfpga*'s "challenge" in this issue and converted his *easy introduction to programming* into Assembly Language.

3.3 Wolfgang's Feedback on Label Alignment

Wolfgang Lenerz also mentioned a foible he has discovered in the way I format my source code for Assembly Language programs and utilities. This is quite a weird foible, in my opinion, but never the less it needs looking into. Here are Wolfgang's comments:

- I downloaded the QL Assembly Language issue 9. As usual, it's fun to read! I have a small comment regarding your label emplacements - I hope you don't mind. Looking at the ql2win filter programs, it starts like this:

```
start
    bra.s checkStack
    dc.l $00
    dc.w $4afb

name
    dc.w name_end-name-2
    dc.b 'QL2WIN'
name_end equ *

version
    dc.w vers_end-version-2
    dc.b 'Version 1.00'
vers_end equ *

checkStack
    ...
```

  Let's suppose I wanted to print out the version, I might start that by using the lines:

```
    lea version,a1
    move.w (a1)+,d1
    ...
```

  somewhere in the code. That would work fine.
  Now make the program name any number of odd characters, like 'QL2WINa' and leave the rest as is. One would think that this shouldn't make any difference.
  Strangely enough though, when you run this, the line `lea version,a1` will give you an address error[3] as A1 will point to an odd address!
  If you now move the `version` label to the line `dc.w vers_end-version-2`, everything will be fine again, whether the name length is odd or even.
  So what happens? The assembler (at least the macro assembler, I don't know about George's) will go through the file and note the address of the version label. This comes right after the bytes of the name, and if the name length is uneven, that address is also odd. Then the

[3] Admittedly, you won't have a problem on *QPC* - but that is because *QPC* emulates an 68020, which can handle word and longword reads/writes at odd addresses. You won't have a problem with *SMSQmulator* either i.e. no address error exception, but you will get "unexpected results"...

> assembler continues and notes the DC.W directive. As this would lie at an odd address, it inserts a filler byte - which comes *after* the version label.
> If you now put the label on the same line as the DC.W directive, the assembler puts in the filler byte before noting the address of version label.
> This is why there is no problem with leaving labels inside of actual code even above the line the label refers to - code is always word aligned. But if you start handling bytes, then you might get an odd address at a label.
> This is why most people leave more space between the beginning of the line and the start of the opcode - it leaves more space for putting the labels in front of them.

Interesting! I'll answer the last point first, I put labels on a separate line because I prefer it that way. I was brought up on COBOL where that was required. It has sort of stuck. I never realised it would cause so much trouble though.

I decided to run a few tests, using *GWASL*, *GWASS* and *QMAC* which are the assemblers I own. The source code was as follows:

```
;     section code

start
    bra.s checkStack
    dc.l $00
    dc.w $4afb

name
    dc.w name_end-name-2
    dc.b 'QL2WIN'
name_end equ *

version
    dc.w vers_end-version-2
    dc.b 'Version 1.00'
vers_end equ *

start2
    bra.s checkStack
    dc.l $00
    dc.w $4afb

name2
    dc.w name_end2-name2-2
    dc.b 'QL2WIN'
name_end2 equ *

version2
    dc.w vers_end2-version2-2
    dc.b 'Version 1.00'
vers_end2 equ *

checkStack
    moveq #0,d0
    rts

;     end
```

I uncommented the `section code` and `end` lines to compile with *QMAC*. The results of my experimenting are shown in table 3.1.

	GWASL	GWASS	QMAC
start	$0000	$0000	$0000
name	$0008	$0008	$0008
name_end	$000D	$000D	$000D
version	$000D	$000D	$000D
vers_end	$001C	$001C	$001C
start2	$001C	$001C	$001C
name2	$0024	$0024	$0024
name_end2	$0029	$0029	$0029
version2	$002A	$002A	$002A
vers_end2	$0038	$0038	$0038

Table 3.1: Weird Label Addresses

The odd addresses, that we care about, are highlighted. Just taking a slightly edited extract from the *GWASL*/*GWASS* listing files, we can see the problem:

```
...
000D 00                         version
000E 000C                           dc.w vers_end-version-2
0010 5665 7273 696F 6E20 312E       dc.b 'Version 1.00'
001C                            vers_end-version-2
```

The label `version`, is at address $000D, or 13 in decimal, which is definitely odd. A similar output can be seen in the listing file from *QMAC*.

Well, that's a bummer indeed! How to resolve the issue? Simples!

```
start
    bra.s checkStack
    dc.l $00
    dc.w $4afb

name
    dc.w name_end-name-2
    dc.b 'QL2WIN'
name_end equ *

  ds.w 0     ; Force word alignment

version
    dc.w vers_end-version-2
    dc.b 'Version 1.00'
vers_end equ *

...

checkStack
    moveq #0,d0
    rts
```

By adding the line ds.w 0 to force word alignment, the problem goes away. This is because the alignment byte is inserted *before* the label and not *at* the label as before. This extract from the *GWASL/GWASS* listing files shows the fix in place:

```
...

000E                          version
000E 000C                         dc.w  vers_end-version-2
0010 5665 7273 696F 6E20 312E     dc.b  'Version 1.00'
001C                          vers_end-version-2
```

The version label is now correctly word aligned and has an even address.

3.4 Wolfgang's Comments on Ql2Win_asm

Wolfgang also sent me a second email, after asking if he could comment on my code, with the following comments and observations.

I'm happy for any comments to be sent to me regarding anything I write, I'm not an expert and sometimes there are other/better ways to do what I do, feel free to point them out!

- I don't understand the reason for the code at label gotLine.

This may be down to my use of IO_FSTRG which I was using originally, instead of IO_FLINE which I changed it to use instead. I think, if I remember, I had a bug or two, or something wasn't quite right with IO_FSTRG hence the change. I might have forgotten to clean up afterwards!

- You test for the presence of LF at the end of the string. But, in your scheme, it will always be there!Well, only if the line was shorter than the buffer length.

Looking at the code (ql2win_asm) I see that gotLine looks for the linefeed, and if not there, skips to putLine to write out the buffer, before reading the next chunk again by branching back to readLoop.

- Indeed, if you get a line that is too long for the buffer, the trap comes back with a buffer full error (-5) (or, sometimes, mistakenly, with an overflow error, -18) - and in that case you abandon the treatment as you leave with the error. So the only time you get to the label is when the line does have the correct LF at the end - no need to check for it.

This is *obviously* (now you mention it) a bug. You are correct, if the buffer is too small I will get an error, so I'll never be able to read the next chunk of the partial line. Thanks for pointing this out.

- You should really also treat the bffl[4] error, by just writing the line so far, and then getting the rest of it.

See above! I thought I was doing this, but the trap handling code breaks it. Duh!

- When there is an EOF at the read trap, you leave w/o error. What if the last file in the file didn't end with LF or CR/LF? You'll also get an EOF, but there will be a remainder still in the buffer. Shouldn't the remainder still be written out?

Yes, another bug that needs fixing. Thanks.

- Consider this code:

[4] Buffer Overflow

```
(jump here from start)
checkStack
          cmpi.w   #numchans,(a7)      ; Two channels is a must
          beq.s    ql2win              ; Ok, skip error bit

bad_parameter
          moveq    #err_bp,d0          ; Guess!
          bra      errorExit           ; Die horribly

ql2win
          moveq    #timeout,d3         ; Timeout
```

After checking for the correct number of channels, you branch to `ql2win` if that is ok, else you continue and treat the error. May I suggest you do the contrary? Check for correct number of channels and branch to the error code if there is a problem, else continue normally. Something like this:

```
bad_parameter
          moveq    #err_bp,d0          ; Guess!
          bra      errorExit           ; Die horribly

(jump here from start)
checkStack
          cmpi.w   #numchans,(a7)      ; Two channels is a must
          bne.s    bad_parameter       ;

ql2win
          moveq    #timeout,d3         ; Timeout
```

The reason is two-fold.
First of all, a branch not taken is faster than a branch taken. So here, you only take the branch if there is a problem with the parameters, and in that case you don't really care if that takes a microsecond more.
Second, you don't interrupt the program flow for whoever is reading your program. Instead of also having to (mentally) jump to the `ql2win` label, you just continue reading. (Admittedly, this last point is more of a personal preference).
The same reasoning could also be applied in the `readloop`, after the trap where you test `D0` for errors - only branch if there is an error, not if there isn't.

All of which makes perfect sense, thanks. I usually try to consider the most often occurring case first, but occasionally, I do things in a different order. I must try harder.

- Maybe you should tell your readers that you don't explicitly close the channels that were opened, since the OS automatically closes all channels opened (or rather owned) by a job when that job is removed.

A good suggestion, thanks. I will. In fact, I'll do it now!

I have omitted to make clear that any open channels, chunks of heap space allocated to a specific job, will be reclaimed by the operating system when the job is removed from the system, either by itself or forcibly by another job. This keeps things nice and tidy and is rather helpful, *however*, there's no harm whatsoever in closing files yourself and returning allocated heap space back to the Common Heap.

- Finally, I presume that you didn't use the SMSQE queue handling (at smsq_ioq_) for didactic reasons for your circular buffer (which is what an SMSQE queue is).

Ah yes, the QDOS/SMSQ/E Queue Handling functions! Marcel mentioned those (on the QL Forum) as well. I was aware of those, but this is some code I've written in C++ for a project, and I will soon (for certain values of soon) be converting it to Atmel AVR Assembly Language for a similar project. I thought I would try it out on a language I'm more familiar with first. It was also a fun project to work with, having been away from the keyboard for a while.

- I know these were a great number of comments, but they wouldn't exist if you hadn't taken it upon yourself to write your greatly enjoyable series!

Comments are *always* welcome, thanks. It's good to hear that I have at least two readers! (Not counting myself, and also, why do I always find bugs/typos after sending out the eMagazine!)

Thanks Wolfgang.

3.5 Bug Fixes for Ql2Win/Win2ql

3.5.1 Ql2Win

The changes to the source code for the bugs identified by Wolfgang are documented below.

To fix the buffer overflow problem identified by Wolfgang, add two extra equates near line 29, which currently reads as:

```
err_bp      equu    -15
err_eof     equ     -10
me          equ     -1
```

Change it to this instead:

```
err_bp      equ     -15
err_eof     equ     -10
err_bffl    equ     -5
err_ovfl    equ     -18
me          equ     -1
```

We have added, in SMSQ/E nomenclature, `err_bffl` and `err_ovfl` as Wolfgang has noted that occasionally, the error returned in `err_ovfl` instead of `err_bffl`. We will use these equates shortly, but first, let's fix the buffer size problem. Around line 83, you'll find this code:

```
q12win
    moveq   #timeout,d3         ; Timeout
    moveq   #buffSize,d4        ; Storage for buffer size for D2
    lea     buffer,a3           ; Start of (write) buffer
```

The problem, identified by Wolfgang, is the `moveq #buffsize,d4` instruction. Any buffer over 127 bytes will be sign extended to a long word. This would make the lower word $FF80 in the case of 128 byte buffers, and as the buffer size is assumed to be positive, this makes for an assumed buffer size of 65,408 bytes rather than the actual 128. As soon as a line longer than 128 is read, bang!

The moveq should be changed to a move.w as in the following:

```
q12win
    moveq   #timeout,d3         ; Timeout
    move.w  #buffSize,d4        ; Storage for buffer size for D2
    lea     buffer,a3           ; Start of (write) buffer
```

Right then, back to the buffer overflow problems. The code starting at line 102 is where we need to test for the two new error codes. It currently looks like this:

```
readLoop
    moveq   #io_fline,d0        ; Fetch lines ending with LF
    move.w  d4,d2               ; Buffer size
    movea.l a4,a0               ; Channel to read
    movea.l a3,a1               ; Read buffer start
    trap    #3                  ; Read a line from input file
    tst.l   d0                  ; OK?
    beq.s   gotLine             ; Yes
    cmpi.l  #ERR_EOF,d0         ; All done yet?
    beq     allDone             ; Yes.
    bra     errorExit           ; Oops!
```

We need to change it to this instead:

```
readLoop
    moveq   #io_fline,d0        ; Fetch lines ending with LF
    move.w  d4,d2               ; Buffer size
    movea.l a4,a0               ; Channel to read
    movea.l a3,a1               ; Read buffer start
    trap    #3                  ; Read a line from input file
    tst.l   d0                  ; OK?
    beq.s   gotLine             ; Yes
    cmpi.l  #ERR_EOF,d0         ; All done yet?
    bne.s   overflow            ; Not yet
    tst.w   d1                  ; Did we read anything?
    bne.s   gotLine             ; Yes, deal with it
    beq     allDone             ; All done now

overflow
    cmpi.l  #ERR_BFFL,d0        ; Buffer overflow?
    beq.s   gotLine             ; Yes, write it out unchanged
    cmpi.l  #ERR_OVFL,d0        ; Buffer overflow (apparently!)
    beq.s   gotLine             ; Yes, write it out unchanged
    bra     errorExit           ; Oops!
```

The observant among us – which usually excludes myself – will notice that I test for any input even when we hit end of file. This helps deal with Wolfgang's over observation that we can lose the last line of the input file if it doesn't fill the buffer.

It will not have a CR/LF added though, if it doesn't have one when read. If it comes without a QL line terminator, it goes out without a Windows one as well. Seems fair to me?

With these changes, and a suitable test file, the code works as good as it did before. All lines, no matter how long, will be correctly written out and converted to Windows format line endings.

I've included a fixed version of the code in the code repository for this issue, so that you don't have to fix it yourself!

3.5.2 Win2ql

As with ql2win, add two extra equates near line 29, which currently reads as:

```
err_bp      equ     -15
err_eof     equ     -10
me          equ     -1
```

Change it to this instead:

```
err_bp      equ     -15
err_eof     equ     -10
err_bffl    equ     -5
err_ovfl    equ     -18
me          equ     -1
```

We have added `err_bffl` and `err_ovfl` as previously. The buffer size problem is next in the source, around line 83, you'll find this code:

```
win2ql
    moveq   #timeout,d3         ; Timeout
    moveq   #buffSize,d4        ; Storage for buffer size for D2
    lea     buffer,a3           ; Start of (write) buffer
```

Once again, the `moveq` should be changed to a `move.w` as in the following:

```
win2ql
    moveq   #timeout,d3         ; Timeout
    move.w  #buffSize,d4        ; Storage for buffer size for D2
    lea     buffer,a3           ; Start of (write) buffer
```

Back to the buffer overflow problems. The code currently starting at line 100 is where we need to test for the two new error codes. It currently looks like this:

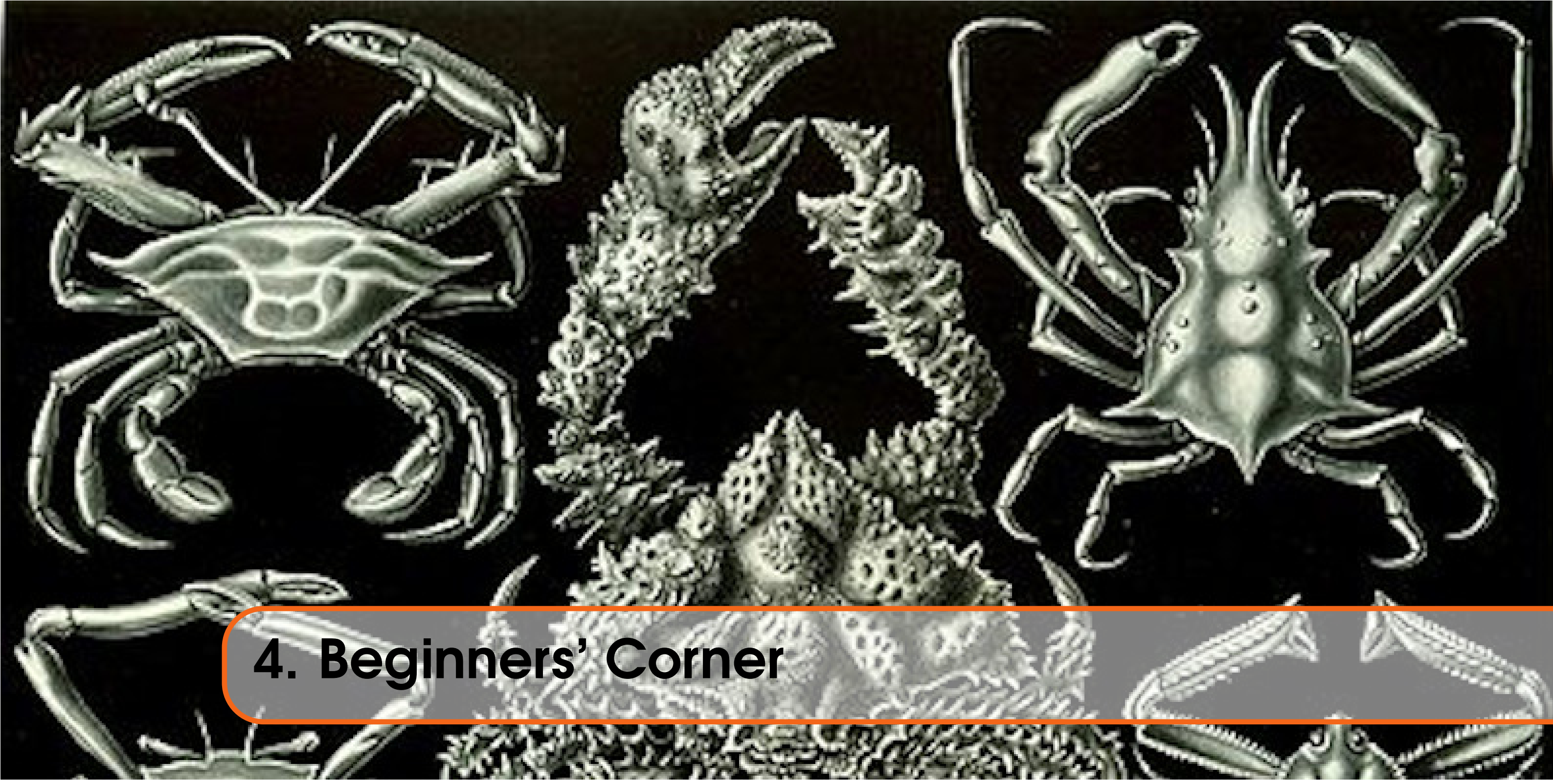
```
readLoop
    moveq   #io_fline,d0        ; Fetch lines ending with LF
    move.w  d4,d2               ; Buffer size
    movea.l a4,a0               ; Channel to read
    movea.l a3,a1               ; Read buffer start
    trap    #3                  ; Read a line from input file
    tst.l   d0                  ; OK?
    beq.s   gotLine             ; Yes
    cmpi.l  #ERR_EOF,d0         ; All done yet?
    beq     allDone             ; Yes.
    bra     errorExit           ; Oops!
```

We need to change it to this instead:

```
readLoop
    moveq    #io_fline,d0        ; Fetch lines ending with LF
    move.w   d4,d2               ; Buffer size
    movea.l  a4,a0               ; Channel to read
    movea.l  a3,a1               ; Read buffer start
    trap     #3                  ; Read a line from input file
    tst.l    d0                  ; OK?
    beq.s    gotLine             ; Yes
    cmpi.l   #ERR_EOF,d0         ; All done yet?
    bne.s    overflow            ; Not yet
    tst.w    d1                  ; Did we read anything?
    bne.s    gotLine             ; Yes, deal with it
    beq      allDone             ; All done now

overflow
    cmpi.l   #ERR_BFFL,d0        ; Buffer overflow?
    beq.s    gotLine             ; Yes, write it out unchanged
    cmpi.l   #ERR_OVFL,d0        ; Buffer overflow (apparently!)
    beq.s    gotLine             ; Yes, write it out unchanged
    bra      errorExit           ; Oops!
```

This is pretty much the same changes as for ql2win. I test for any input even when we hit end of file, as before, so that when the last line of a file doesn't fill the buffer it will still be written out.

As with ql2win, it will not have a QL line ending LF added if there wasn't one presnet when the line was read. If it comes without a Windows line terminator, it goes out without a QL one as well. Seems fair to me?



4. Beginners' Corner

4.1 Introduction

This is a new feature starting in this issue. It stems from a post on the QL Forum from *TMD2003* who was wondering about how to get started learning Assembly Language as a “noob”. The topic is this one[1] on the forum. A number of useful answers were given, some pointing at my book and these eMagazines.

4.2 Do Basic Things

On the second page of the thread, *Tinyfpga* issued this “sort of” challenge:

In Basic all I have to do start my journey into the programming world is to:- Type into QD

```
OPEN #1,con_
OUTLN #1,310,60,50,300
CLS #1 : BORDER #1,1,7
INPUT #1,a$
```

I then save the text in a ram disk, press the execute button and ,"hey presto", I see an app on my screen. What could be an easier introduction to programming?

Well, I like a challenge, even when it’s not really intended as one, so I decided I would create the code to do the necessary.

The first thing to note is how simple it looks from SuperBASIC to do what appears to be a simple thing. It’s 4 lines of code, how hard can that be?

Well, SuperBASIC takes the typed in commands, parses them and if all is well, execution takes the

[1] https://qlforum.co.uk/viewtopic.php?f=3&t=3976

tokenised code and converts it to various calls to the ROM, Toolkits and such like, and the result is a seemingly simply result. Under the covers there's a whole lot of work taking place.

4.2.1 Program Constants

I created the code in the various listings in this chapter to do the conversion from SuperBASIC to Assembly. I chose to create a job, that multitasks alongside SuperBASIC but I could have made it a CALLable routine instead. Let's dive in! Listing 4.1 shows the constants I used.

In the original, I had QDOS versions for the vectors and traps, for this eMagazine, I've converted things to use SMSQ/E versions.

```
;=================================================================
; A simple multi-tasking job for TinyFPGA/TMD2003 to:
;
; Open #n,con_
; OUTLN #n,310,60,50,300
; CLS #n
; BORDER #n,1,7
; INPUT #n, some_text_from_user
; Die!
;=================================================================
; Norman Dunbar
; 20 November 2021.
;=================================================================


;-----------------------------------------------------------------
; Some definitions to make life simple(r)!
;-----------------------------------------------------------------
WHITE       equ 7                   ; White colour
BLACK       equ 0                   ; Black colour

BORDCOLOUR  equ WHITE               ; Border colour
BORDWIDTH   equ 1                   ; Border width
PAPER       equ BLACK               ; Paper colour
INK         equ WHITE               ; Text colour

CON_W       equ 310                 ; Console width
CON_H       equ 60                  ;    "    height
CON_X       equ 50                  ;    "    X position
CON_Y       equ 300                 ;    "    Y position

BUFFERSIZE  equ 256                 ; User input buffer size

;-----------------------------------------------------------------
; I use GWASS as my assembler, it has the QDOS traps etc built
; in. It doesn't however, have the PE stuff, so these two are
; required.
;-----------------------------------------------------------------
IOP_PINF    equ $70                 ; Get PE information
IOP_OUTL    equ $7a                 ; OUTLN

;-----------------------------------------------------------------
; GWASS also doesn't know about SMSQ/E names, so these are now
; required.
;-----------------------------------------------------------------
```

```
OPW.CON     equ $c6             ; Open a console, border, etc
IOW.CLRA    equ $20             ; CLS whole window
IOB.SBYT    equ $05             ; Print one byte to channel
IOW.ECUR    equ $0E             ; Enable cursor
IOB.FLIN    equ $02             ; Fetch a line of text plus LF
UT.WTEXT    equ $d0             ; Print some text
SMS.SSJB    equ $08             ; Suspend a job
SMS.FRJB    equ $05             ; Force remove a job
```

Listing 4.1: Tinyfpga - Constant Definitions

There's nothing much here apart from the new style, SMSQ/E names. Most of these are trap codes but a couple are vectored routines which can be used to call the trap routines but in a simpler manner. For certain values of simpler, sometimes.

4.2.2 Job Header

Listing 4.2 shows the "pretty much boilerplate" code that all jobs need to have at the start. Seasoned readers can skip this next explanation!

A job in SMSQ/E requires a standard job header. This is the same as it was back in the old QDOS days, and consists of 10 bytes of boiler plate code, followed by the job's name in the standard format of a word defining the length of the name, followed by the bytes of the name.

There are numerous ways to set up the first 6 bytes. I prefer a short branch to the job's actual start followed by a long word of zero. Back in the days when I wrote code for a living, this could have been used to set up a serial number for copies of the program(s) - there's enough room in a long word for 2^{32} different values. Minus 1 if you don't want a serial number of zero!

Regardless of how you set up the first 6 bytes, bytes 6 and 7 (starting from zero) will always be the constant value of $4AFB. This is the marker word used by QDOS and SMSQ/E to indicate a job's code follows.

Immediately after the marker word, we have a word defining the size of this job's name, here we see it is 8 bytes long, followed by the bytes of the job name itself. In this case, I set the job name to "TinyFPGA".

```
;------------------------------------------------------------------
; Job's require a header. This is basically boilerplate, except
; for the job name's length and the name of the job. This will
; need to be EXEC/EXEC_W/EX or EW'd to execute it.
;------------------------------------------------------------------
Start
        bra.s open_console      ; Skip over job header
        dc.l 0                  ; 4 bytes, can be any value
        dc.w $4afb              ; Job flag, must be $4afb
        dc.w 8                  ; Length of job name
        dc.b "TinyFPGA"         ; Bytes of job name
```

Listing 4.2: Tinyfpga - Job Header

Why does a job need a name? It doesn't! But if it has one, it makes life easier when using the SuperBASIC JOBS command, or the QPAC2 Jobs Thing to list the various jobs running in the system.

4.2.3 Opening a Console Channel

The first task in the challenge is to open a console channel. How we do this is shown in Listing 4.3. Bear in mind that there are other Trap calls available to the programmer to open files, and a console channel is just a file. Here we will use a vectored utility which allows the "open channel" to be simplified to one call. Without the vector we would have to open the channel, set the paper, strip and ink colours, and set the border width and colour too. I don't know about you, but I prefer to do less typing in my programs!

The 4 bytes at label `con_def` define the border, paper and ink attributes. Here I'm using some of the constants defined in Listing 4.1 which makes it easier if I decide I don't like *Tinyfpga*'s choice of colours, and want to change things.

The 4 words immediately following, at label `outln_def`, are used for two separate purposes. The first is when we open the console channel – they define the width, height, x position and y position of the opened console channel.

The second use is when we try to OUTLN the channel. That function requires a 4 word block of data in exactly this format, so we can use the same code for two different things.

The fact that label exists between the two lumps[2] of code makes no difference. A label doesn't generate any code or data in the assembled program.

```
;------------------------------------------------------------------------
; Needs a channel open first. This can be done in a couple of
; ways, but this is probably the easiest. It will open a con_
; channel of the required size and border it.
;
; UT_CON uses all of the following from con_def and outln_def
; IOP_OUTLN only uses the latter.
;------------------------------------------------------------------------
con_def
        dc.b BORDCOLOUR                 ; Border colour
        dc.b BORDWIDTH                  ; And width
        dc.b PAPER                      ; Paper/strip colour
        dc.b INK                        ; Ink colour

outln_def
        dc.w CON_W                      ; Width
        dc.w CON_H                      ; Height
        dc.w CON_X                      ; X pos
        dc.w CON_Y                      ; Y pos

open_console
        lea con_def,a1                  ; Parameters
        move.w OPW.CON,a2               ; C6 = CONSOLE required
        jsr (a2)                        ; Open Console & set params
        bne die                         ; If it failed - bale out
```

Listing 4.3: Tinyfpga - Open a Console

Having got the console channel definitions out of the way, we can open it and set the attributes with a call to the vectored `OPW.CON` utility. This call requires that the `A1.L` register points at the byte defining the border colour, thus `con_def`, and that's all.

[2] This is a technical term!

To call a vectored utility we get a word from the ROM (as was in the old days) at a certain address, in SMSQ/E this address is labelled `OPW.CON` and this contains the address where the actual code we wish to execute lives. In the ROM, there is a long list of available vectored routines which we can use in our own code.

After calling the vectored utility, we will have an error code in `D0.L` and, if nothing went wrong, `D0.L` will be zero, the `Z` flag will be set for us, and `A0.L` will contain the channel identifier for the newly opened channel. Note that this is not the same as a SuperBASIC channel number. SuperBASIC holds a table of channel identifiers and indexes that table using the channel number, not the actual channel identifier.

If there was an error, the error code is in `D0.L` as noted, and the `Z` flag will not be set. In this case, we simply jump to the code at label `die`, where the error code will be returned to SuperBASIC and the job aborted.

4.2.4 Do We Have the Pointer Environment?

Tinyfpga's challenge was to `OUTLN` the opened channel. To do this we need to ensure that the Pointer Environment is present. The code in Listing 4.4 does exactly this by calling the `IOP.PINF` function. This is not a vectored utility, this is a trap call. These are slightly different as they execute as *exceptions* and not as normal subroutuines.

To test if the PE is present we need a channel identifier in register `A0.L` – we already have that from above; we need a timeout in register `D3.W` and we need the value $70 in register `D0.L`[3]. After setting up the registers, a `trap #3` call is made and on return, an error code will be found in register `D0.L` however, the `Z` flag will not be set.

```
;----------------------------------------------------------------------
; We only get here if it worked. A0 now holds the channel id.
; Most, if not all, QDOSMSQ code preserves the A0 register.
;----------------------------------------------------------------------
; Check if the PE is installed. If not, ignore the error and
; skip to handle clearing the screen "manually".
;----------------------------------------------------------------------
check_pe
          moveq  #IOP.PINF,d0          ; IOP_PINF
          moveq  #-1,d3                ; Timeout (Preserved)
          trap   #3
          tst.l  d0                    ; Errors?
          bne.w  cls_console           ; PE missing
```

Listing 4.4: Tinyfpga - Check for Pointer Environment

Why is the `Z` flag not set? Because the trap code executes as an exception, part of what it does before executing is to stack the status register. After execution, the exception code returns to user code using the `RTE` instruction, not `RTS`. The `RTE` unstacks the old status register value and puts it back into the status register. This obliterates any flags set within the exception (ie, `trap`) code so none of the flags will represent what happened within the trap code.

This means that every time we return from a trap call, we must test if `D0.L` is zero or not. This will set the `Z` flag accordingly and we can then tell if the `trap` call worked or failed.

In this case, we don't really care if the PE is present or not. Well, I assumed this to be the case based on the challenge. If the PE is found, we can `OUTLN` the channel as desired, but if it wasn't

[3] It need not be actually the *whole* of register `D0`, however, a `moveq` instruction fills the whole of the register.

found, we can just CLS the channel anyway and carry one – in this example.

4.2.5 Pointer Environment Found

If the PE is present, we can OUTLN the channel. Listing 4.5 shows the code to do this. Again, IOP.OUTL is a trap call, not a vector. We are required to set register D0 to $7A – which is what IOP.OUTL is defined as; Register D1.L should hold the X and Y shadow widths for the OUTLN – we are not using shadows; Register D2[4] should hold 1 to preserve the window contents so that a previously OUTLN'd window can be moved and the contents preserved, or zero to not bother. As this is the first OUTLN call for this window, we have to use zero.

```
;------------------------------------------------------------------------
; PE is present.
;------------------------------------------------------------------------
; OUTLN the window,D3 (timeout) was preserved in IOP_PINF as
; was A0 (channel id for the console).
;------------------------------------------------------------------------
outln_console
        moveq   #IOP.OUTL,d0        ; IOP_OUTL
        moveq   #0,d1               ; No shadows
        moveq   #0,d2               ; Don't preserve contents
        lea     outln_def,a1        ; Window sizes (W,H,X,Y)
        trap    #3                  ; Do it
        tst.l   d0                  ; Errors?
        bne     die                 ; Yes, bale out
```

Listing 4.5: Tinyfpga - Pointer Environment Present

After the trap call, we check for errors as explained above, and if there were any, we exit the job via the code at label die. If there were no errors, we drop in to the following code to clear the screen.

4.2.6 Clear Screen

If the PE was found to be missing, we don't really care in this small example as all we need from the PE is the OUTLN call, and we drop in here to clear the console channel. If the PE was present, the channel has been OUTLN'd, but we are here again to clear it also. Listing 4.6 sets D0.L to $20 also known as IOW_CLRA; D3.W is still the same timeout as before; A0.L is still the console channel identifier. Those are all we need, so we then trap #3 to clear the screen. On return, D0.L is tested in the usual manner and on any errors, we bale out of the job.

```
;------------------------------------------------------------------------
; PE is missing.
;------------------------------------------------------------------------
; Clear the screen. The timeout and channel ID have been
; preserved over the last two routines.
;------------------------------------------------------------------------
cls_console
        moveq   #IOW.CLRA,d0        ; CLS whole window
        trap    #3                  ; Do it
        tst.l   d0                  ; Any errors, probably not
        bne     die                 ; Yes, bale out
```

Listing 4.6: Tinyfpga - Clear the Screen

[4] The documentation doesn't mention a size!

4.2.7 Print a Prompt

This was not part of the original challenge, but I added it when debugging a problem. I decided to leave it in. The code in Listing 4.7 simply prints a prompt of '>' to the console channel. This is facilitated using IOB.SBYT which requires that D0.L is set to $05, aka IOB.SBYT one of our constants; D1.B is set to the character to be sent to the channel; D3.W is the same old timeout value and A0.L is the channel identifier.

After the trap, D0.L is tested in the usual manner and we exit the job if any errors occurred.

```
;------------------------------------------------------------------------
; Print a ">" prompt to the channel. Timeout/channel Id still
; preserved.
;------------------------------------------------------------------------
con_prompt
        moveq   #IOB.SBYT,d0        ; Print one character
        moveq   #'>',d1             ; The prompt character
        trap    #3                  ; Do it
        tst.l   d0                  ; Any errors?
        bne     die                 ; Yes, bale out
```

Listing 4.7: Tinyfpga - Print a Prompt

4.2.8 Enable the Cursor

This is also not part of the original channel, but a cursor must be enabled when we want to get input from a console channel. We have yet another trap call to help with this. The code in D0.L is IOW.ECUR the timeout and channel identifier are in the usual registers. Errors are tested in the normal manner as this is a trap call, not a vectored call.

Listing 4.8 shows the code.

```
;------------------------------------------------------------------------
; Enable the channel's cursor. We need one to get input from
; a console channel. Timeout & channel id preserved still.
;------------------------------------------------------------------------
con_cursor
        moveq   #IOW.ECUR,d0        ; Enable cursor
        trap    #3                  ; Do it
        tst.l   d0                  ; Errors?
        bne     die                 ; Yes, bale out
```

Listing 4.8: Tinyfpga - Enable a Cursor

4.2.9 Get Some Input

Listing 4.9 shows the code we need to obtain a line of input from the user. How long is a line? Well, that's all down to the programmer. However, we define an input buffer, at label input_buffer, to be 256 bytes long, plus an extra 2 bytes. BUFFERSIZE is one of our constants and defaults to 256, but you can change it.

Why 2 extra bytes? We are accepting a string from the user. Strings in SMSQ/E are defined as a word holding the size followed by the bytes of the string – the job name in this code, for example. If we wanted to process the string in some way, we would need to know the length.

IOB.FLIN is the trap call we need to use. This will have D0.L holding $02; D2.W holding the maximum buffer size we will allow; A1.L points at the destination for the input we receive and D3.W and A0.L are the usual timeout and channel identifier.

You will note A1.L is pointing at the third byte in the input buffer, this is where the text will be stored, the first two bytes are used for the size of the text we obtained.

After the trap call, if there are no errors, we reset A1.L to the start of the buffer this time – it was changed by the trap call – and store the size word there. We now have a proper SMSQ/E string. The size word in this case comes from D1.W on return from the trap, where it holds the size of the input received, *including* the terminating linefeed.

What happens if there was more input than the buffer size? Nothing, the buffer will be filled to capacity and the trap call will return. The last character in the buffer will not, therefore, be a linefeed in this case.

```
;------------------------------------------------------------------------------
; Grab some input from the console. The timeout and channel id
; are still valid. We point A1 at input_buffer+2 as we need the
; start of the buffer to hold the length of the text that
; follows.
;------------------------------------------------------------------------------
get_input
        moveq #IOB.FLIN,d0         ; Fetch input with Linefeed
        move.w #BUFFERSIZE,d2      ; How big is my buffer?
        lea input_buffer+2,a1      ; Input buffer space
        trap #3                    ; Fetch input D1.W = size
        tst.l d0                   ; Errors?
        bne die                    ; Fraid so

        lea input_buffer,a1        ; The buffer start this time
        move.w d1,(a1)             ; Store the input size (inc LF)
        bra print_input            ; Skip over inopt buffer

input_buffer
        ds.w    BUFFERSIZE+2       ; Buffer for data input
```

Listing 4.9: Tinyfpga - Request Input

Setting A1.L to the start of the buffer sets us up nicely for printing out the received text.

4.2.10 Printing the Text

This wasn't part of the challenge, but I added it to show that the data we typed in to the channel was in fact well received and correctly saved. Listing 4.10 shows the code we use to print out the received input text. This will include the trailing linefeed is one was present.

This code uses a vectored utility, UT.WTEXT, which internally calls another trap #3 function to print out the text. Why have I used the vector? The vector requires a pointer to an SMSQ/E string in A1.L – we already have that. It requires a channel identifier and timeout in A0.L and D3.W – we already have that.

The corresponding trap call needs the string length in D1.W and A1.L pointing to the bytes of the text. I prefer using UT.WTEXT.

After the call, we know that the Z flag is set if no errors occurred, so there's no need to test D0.L on return. If errors are detected, we again exit the job.

Note, this vectored code destroys the timeout value in D3.W. However, at this point we are done with the infinite timeout we have been using.

```
;------------------------------------------------------------------
; Print the input that the user gave us, including the line
; feed at the end. A1 points to the text's word size, D3 will
; be corrupted by this vector call (timeout) but the channel
; id in A0 will not.
;------------------------------------------------------------------
print_input
        move.w UT.WTEXT,a2      ; Print a string of bytes
        jsr (a2)                ; Print it
        bne.s die               ; Ooops, error
```

Listing 4.10: Tinyfpga - Printing the Input

4.2.11 Hang on a Few Seconds!

Ok, we are done. Except to give the user a chance to see the text printed on the channel, I've added yet another extra to the challenge code. The currently running job, named "Tinyfpga", will be suspended for a couple of seconds. Listing 4.11 shows how this is done.

SMS.SSJB is a trap call which suspends a job from execution for the number of frames specified in the D3.W register. D1.L holds the job identifier, or -1 for the current job; and A1.L points at a byte which will be cleared when this job resumes. As we have no need to signal our reappearance, we use zero.

Note that the number of frames is 200. This is 4 seconds in the UK and countries with a 50 Hz mains frequency. In the USA it's 60 Hz, so in the USA the delay will be $3\frac{1}{3}$ seconds.

After the trap, we do not test for errors, we are about to die anyway.

```
;------------------------------------------------------------------
; Suspend the job for a couple of second to let the user see
; the output. Then die. Will corrupt A0 but who cares!
;------------------------------------------------------------------
suspend_job
        moveq #SMS.SSJB,d0      ; Suspend a job
        moveq #-1,d1            ; This job
        move.w #200,d3          ; 4 seconds is 200 frames
        movea.l #0,a1           ; No byte to be cleared
        trap #1                 ; Suspend the job
```

Listing 4.11: Tinyfpga - Delay Before Ending

4.2.12 Death of a Job

The job is now complete. We are not required to loop around and keep running, so we cannot allow the job's code to simply stop, we need to remove the job from the system. Listing 4.12 shows how to force remove a job using the SMS.FRJB trap call.

The error code in D0.L is copied into D3.L for return to SuperBASIC – more on that soon – and the job identifier is loaded into D1.L, we are using -1 again to indicate the current job. After the trap, no code will be executed as the job *is no more, it has shuffled off its mortal coil and gone to meet its maker!*[5]

[5] From Monty Python's *Dead Parrot* sketch.

```
;------------------------------------------------------------------
; The job is complete, remove it from the system. Any error
; codes in D0 are copied to D3 ready for EXEC_W/EW to collect.
; EXEC/EX don't bother.
;------------------------------------------------------------------
die
        move.l  d0,d3               ; Any errors?
        moveq   #SMS.FRJB,d0        ; Force Remove a job
        moveq   #-1,d1              ; -1 means "this job"
        trap    #1                  ; Kill this job
```

Listing 4.12: Tinyfpga - Death of a Job

4.2.13 Error Codes

When the job is started using `EXEC`, or `EW`, it will never tell you how it ended. There could have been errors at any stage and you will never know about it. Why not? Because `EXEC/EW` start up a job and then return. These commands do not wait for the job to complete. How then can they be expected to be able to obtain the job's error code when it finishes – it might run for days after all.

During testing, when the code wasn't crashing, I ran it with `EXEC_W` or `EW`. These commands wait for the job to complete before returning to SuperBASIC. In this case, the job's error codes can be returned to SuperBASIC.

4.3 Assembling the Code

If I were to assume that you have downloaded GWASS[6] for QPC2 and other 68020 based emulators, or GWASL[7] for the QL and 68008 based emulators, and have the code saved as `ram1_Tinyfpga_asm`, then assembling the code is as simple as this:

- `EXEC gwass60_bin` or `EXEC gwasl_bin` to start the assembler;
- Select the option to start assembling;
- Type in the filename: `ram1_Tinyfpga_asm`
- Wait.

After a successful assemble, `ram1_Tinyfpga_bin` will be the executable job. To run it:

- `EXEC ram1_Tinyfpga_bin: REMark Alternatively, EX ram1_Tinyfpga_bin`

On a successful execution, a small window will open, with black paper, white ink and a white, one pixel border. A '>' prompt will be displayed in the top left corner. Type some text and press ENTER. The text you typed will be printed, the job will pause for 4 seconds, and then vanish.

4.4 Summary

So that's the Assembly Language version of *Tinyfpga*'s challenge. There's a lot going on under the covers of SuperBASIC that programmers almost never see. When you start delving ito Assembly Language, you are responsible for just about everything! Thankfully, SMSQ/E provides numerous utilities and features that you can call upon to make life easier.

[6] http://www.dilwyn.me.uk/asm/gwassp22.zip

[7] http://www.dilwyn.me.uk/asm/gwaslp08.zip

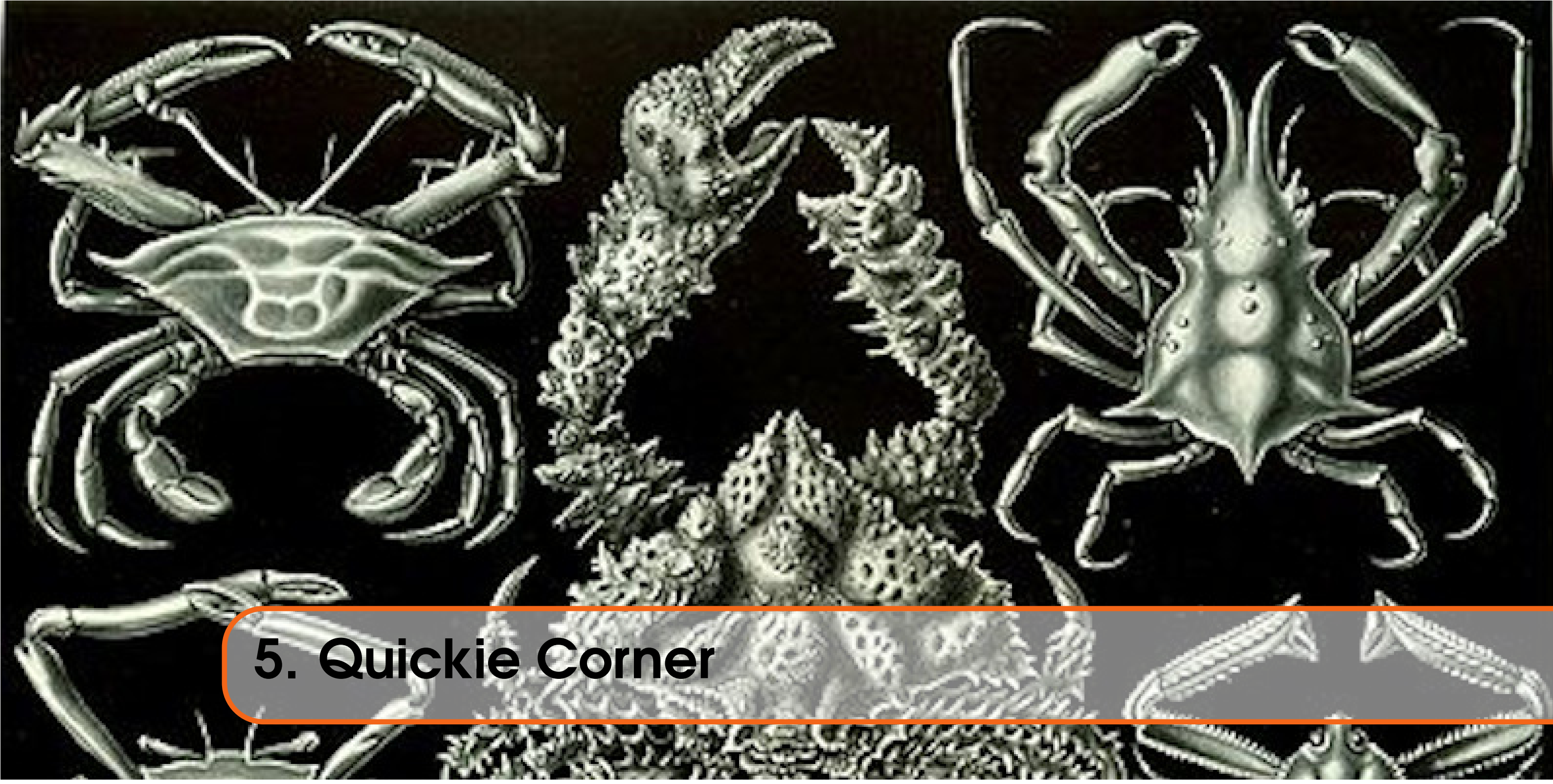
In future issues, I'll be delving into a few more of these with, hopefully, enough explanation for beginners to get started with.

Get hold of the SMSQ/E Reference Manual from:

- Here[8] for the PDF version; or
- Here[9] for the ODT version.

[8] http://www.dilwyn.me.uk/docs/manuals/QDOS_SMS%20Reference%20Guide%20v4.5.pdf

[9] http://www.dilwyn.me.uk/docs/manuals/QDOS_SMS%20Reference%20Guide%20v4.5.odt



5. Quickie Corner

5.1 Speedy Stuff

What's the fastest way to clear a data register from whatever value it has, to zero? `Moveq #0,Dn` do you think? Maybe `clr.l Dn`? Let's see. The timings are taken from the Motorola Semiconductor's MC68000 Programmers Reference Guide, 4th Edition, Appendix E.

Table 5.1 shows a few instructions which can clear a data register and their timings.

Instruction	Clock Cycles
`Move.l #0,Dn`	12
`Moveq #0,Dn`	**4**
`Sub.l Dn,Dn`	8
`Eor.l Dn,Dn`	8
`Clr.l Dn`	6
`Andi.l #0,Dn`	16
`Move.l Dx,Dn`	**4**
`Move.l addr.W,Dn`	16
`Move.l addr.L,Dn`	20
Exg Rn,Dn	6

Table 5.1: Clearing a data register

It appears that two instructions are quickest. `Moveq`, as its name suggests but also moving data from one register to another, assuming of course that the source register is indeed already cleared. Now, what about address registers?

Table 5.2 shows a few instructions which can clear an address register and their timings.

It would appear that clearing an address register is quickest when you have another register already holding zero, but if not, subtracting the register from itself is the next quickest option.

Instruction	**Clock Cycles**
Exg Rn,Dn	6
Movea.l #0,An	12
Suba An,An	6
`Move.l Rn,An`	**4**
`Move.l addr.W,An`	16
`Move.l addr.L,An`	20

Table 5.2: Clearing an address register

6. Free Pascal Compiler

Back in November 2020, there was a "thing" called *QLvember*[1] and someone – Károly Balogh is his name, he's from Hungary, and *Chainq* is his nickname on the QL Forum – decided to try and make the Free Pascal Compiler (FPC) work on the QL. After an exchange of information on QL Forum, this thread[2], he went off and created a cross compiler for the QL.

Eventually, myself and Marcel got slightly involved and helped create a starting set of Pascal "units" (aka libraries if you wish) to make writing Pascal easy for QL users.

Then I got distracted, my wife and I, after 25 years of marriage (almost), heard the pitter patter of tiny feet! We bought a puppy! Looking after one of those is hard work – I'd forgotten how hard it can be – but he's settled down a bit better now, so I'm getting back into development again. Just as well I wrote a document detailing how to get hold of the development tools and source code required to build the compiler and the QL Units. You can find it on my GitHub page, here[3].

This article is an attempt to get more people interested enough to help out writing code for the QL's run time library and other units. If we can do this, we will have a proper modern, frequently updated cross compiler for the QL. I'm running FPC on Linux, but it's also available for Windows too. Either way, we can now cross compile Pascal code for the QL – there are restrictions obviously, because the RTL and Units are incomplete, but if more of us join in, we might get something done.

Dare I mention, the FPC compiler is able to compile Object Oriented code, which runs on the QL. I have a post on the forum here[4] about this. Tony Tebby will not be amused!

[1] I really hate it when they do stuff like that, taking a month and a concept, and putting them together. Things like Dryanuary, Veganuary, QLvemver – it makes my teeth crawl! Anyway, enough from the old git!

[2] https://qlforum.co.uk/viewtopic.php?f=3&t=3057&p=37465#p37465

[3] https://github.com/NormanDunbar/FPC-CrossCompiler-QL/releases/latest/.

[4] https://qlforum.co.uk/viewtopic.php?f=3&t=3725

6.1 Writing Code

Basically, I'm writing QDOS and/or SMSQ routines to match those documented in the Technical Guides. For example, Listing 6.1 is the Pascal `FileOpen` function, which opens a file, this is found in the `sysutils` unit for the QL.

```
function FileOpen(const FileName: rawbytestring; Mode: Integer):
    ⟹ THandle;
var
  QLMode: Integer;
begin
  FileOpen:=-1;
  case Mode of
    fmOpenRead: QLMode := Q_OPEN_IN;
    fmOpenWrite: QLMode :=  Q_OPEN_OVER;
    fmOpenReadWrite: QLMode := Q_OPEN;
  end;
  FileOpen := io_open(pchar(Filename), QLMode);
  if FileOpen < 0 then
    FileOpen:=-1;
end;


function FileGetDate(Handle: THandle) : Int64;
begin
  result:=-1;
end;
```

Listing 6.1: FPC FileOpen function

You can see, on line 11, about half way down, there's a call to `io_open`, passing the filename and the mode the file is to be opened. The code for `io_open` is shown in Listing 6.2, and is found in the `qdos.inc` file. You will hopefully notice from the code that there are two io_open functions:

- `io_open_qlstr`: which opens a file using a QL style string for the file name - a word count followed by the bytes of the string.
- `io_open`: Which is called from Pascal, and converts the Pascal parameters passed, to suitable ones to call `io_open_qlstr`.

You might also notice a function called `FileGetDate` in Listing 6.1? That one simply returns -1 because it has yet to be written for the QL. And this is a gentle reminder that we need more people to get involved.

```
const
  _IO_OPEN = $01;
...

function io_open_qlstr(name_qlstr: pointer; mode: longint): Tchanid;
    ⟹ assembler; nostackframe; public name '_io_open_qlstr';
asm
  movem.l d2-d3,-(sp)
  move.l name_qlstr,a0
  moveq.l #-1,d1
  move.l mode,d3
  moveq.l #_IO_OPEN,d0
  trap #2
  tst.l d0
  bne.s  @quit
  move.l  a0,d0
@quit:
  movem.l  (sp)+,d2-d3
end;

function io_open(name: pchar; mode: longint): Tchanid; public name '
    ⟹ _io_open';
var
  len: longint;
  name_qlstr: array[0..63] of char;
begin
  len:=length(name);
  if len > length(name_qlstr)-2 then
    len:=length(name_qlstr)-2;

  PWord(@name_qlstr)[0]:=len;
  Move(name^,name_qlstr[2],len);

  result:=io_open_qlstr(@name_qlstr,mode);
end;
```

Listing 6.2: FPC Io_open function

6.1.1 Parameter Passing

The default method of passing parameters around in Free Pascal is named *register*. This method passes parameters using registers where it can, as follows:

- The first ordinal number (ie, a numeric value) is passed in register `D0`.
- The second ordinal number (ie, a numeric value) is passed in register `D1`.
- The first pointer or reference is passed in register `A0`.
- The second pointer or reference is passed in register `A1`.
- Any remaining parameters are passed on the stack from left to right.

If you look closely at `io_open` you will see that it takes the address of the QL String for the filename as a `pointer` data type. This will be passed in register `A0` as per the convention, however, the eagle-eyed among you will have spotted that there is an instruction `move.l name_qlstr,a0` which, if the above information is correct, simply copies `A0` to `A0`.

This is indeed correct, however, it *should not* be deleted as a spurious line of code. There are other parameter passing modes, documented on the Free Pascal Wiki[5], which use a different manner of passing parameters. If you delete that *spurious* line, and the mode changes in future, code will stop working.

Another problem that might bite you, is when you are usually a bit anally retentive[6] about setting up the registers in order, you might, as I did, find yourself overwriting registers `D0` and `D1` with QDOSSMSQ trap settings, for example, *before* you've grabbed the parameters passed from Pascal. How do I know this? Don't ask – but it took a lot of `QMON`ing to find out what was going wrong!

Returning Results

- Return values which are 32 bits in size, or smaller numeric values are returned in register `D0`.

[5] https://wiki.freepascal.org/m68k#Registers

[6] Another technical term!

- Pointers are returned in register `A0` or `D0` (platform and calling convention specific).
- 64 bit ordinal values are returned in register pair `D0/D1`.

You can see, in Listing 6.2, the channel ID being returned from `io_open_qlstr` in `D0` instead of the usual QDOSMSQ manner of returning it in `A0`.

6.2 Join In If You Can

If you can write assembly code, then why not join in? Grab my document and have a go at making one or more of the missing functions for the QL, work. It's fun and a good way to be useful – although my wife, hello dear, begs to differ on my definition of *useful*!

The main forum thread starts from *chainq*'s post, linked above. That's where I have been putting my patch files and they've been accepted into the FPC project, some slightly amended by *chainq*, which is fine, he knows this stuff, I'm just a code monkey! There's also a good deal of information on that thread about the way that FPC works for the QL.

If you have questions, ask away, someone will chime in and hopefully, help you out.

One more thing, the FPC project recently[7] (8th August 2020) migrated everything from Subversion as a version control system, to git. The project is now hosted on GitLab and mirrored to gitHub, for safety, but GitLab is the main repository. You can find it here[8].

[7] Depending on when you read this, maybe not quite so recently!

[8] https://gitlab.com/freepascal.org/fpc/source.git

7. Heaps

Note: Unless otherwise noted in the text, code listings in this chapter are fragments only, they do not make up a full, working application.

There are two kinds of heap in the QL:

- The *Common Heap*.
- *User Heap(s)*.

What's the difference? Well, details are below but in summary:

- Allocation requests in the common heap will have a 16 byte overhead added to the space requested, and the amount requested will be rounded up to a multiple of 16. However, if this leaves a 16 byte gap, then the value will be rounded up to a multiple of 32 instead. **Warning**: This is an implementation detail and you should not rely on the rounding being 16 or 32. Check `D1` on return and make sure you got what you requested[1];
- The value returned in `D1`, the amount of space allocated, *includes* the 16 byte overhead plus the rounding;
- The 16 byte overhead is prior to the address returned in `A0` for the base of the space allocated;
- When deallocating common heap, you don't need to remember the size that was allocated;
- Common heap addresses are absolute;
- User heap space requests are rounded to multiples of 8 bytes;
- There is an 8 byte overhead, but this is *not added* to the size requested;
- The overhead is at the two long words pointed to by the address returned in `A0` for the base of the area allocated;
- When deallocating user heap space, you need to remember the size of the space allocated.
- User heap addresses are relative to `A6`.

[1] Thanks Marcel.

7.1 Common Heap

The common heap is an area of RAM, allocated from the free space area in the memory map, between the addresses pointed to by `SYS_CHPB` – the base of the common heap area – and `SYS_FSBB` – the base of the free memory area – both relative to `A6` of course! The common heap is mainly used by QDOS/SMSQ to hold such things as channel definition blocks and other bits of working storage required by various drivers. Individual jobs can also request areas of space in the common heap for their own use.

- When a channel is closed the space allocated in the common heap will be reclaimed automatically;
- Likewise, if a driver is (able to be) removed from the system, it's working storage space will be reclaimed;
- When a job is removed from the system, perhaps forcibly, then any areas of common heap owned by the job are also reclaimed by the system.

All of the above can also "voluntarily" free up any space obtained in the common heap, when it is done with.

To try to avoid fragmenting the common heap, if is advisable to free up space in the opposite order from that in which is was allocated. For example, if a job requests 10 Kb, 5 Kb then 30 Kb from the common heap, it should, but it's not mandatory, to free the 30 Kb allocation, then the 5 Kb allocation and finally, the 10 Kb allocation. This doesn't work all the time – some other jobs may have allocated space between those mentioned and freeing will still leave the common heap a little more fragmented than is desirable.

Note: There are similar problems with heap allocation on other operating systems, not just the QL. The advice on those systems is also, deallocate in the opposite order that you allocated.

A job should, really, request a single chunk of common heap and use that as a user heap, allocating space from the user heap rather than the common heap, especially if the allocations are small in size[2] as this will help to reduce common heap fragmentation. User heaps are discussed in the next section.

Fragmentation of the common heap can lead to situations such like a channel which cannot be opened as there isn't enough *contiguous* free space to create a channel definition block.

Note that when allocating common heap space, the amount of RAM allocated will be rounded up to a multiple of 16 (or, potentially, 32) bytes. Each chunk of common heap allocated will have a 16 byte overhead on top of the space requested. For example:

- The code requests 10 bytes of common heap;
- The allocation will be 16 bytes after rounding up;
- The total space allocated from the common heap's free space will be 32 to include the 16 byte overhead;
- The value returned in `D1`, the amount of space allocated, will include the overhead and rounding – it will be 32.

The 16 byte overhead holds the data detailed in Table 7.1.

The long word at offset -$0C will normally be zero for space allocated by jobs – not drivers – as the free space list in the common heap is maintained by a separate linked list of pointers, based on the system variable `SYS_FSBB`.

[2] Based on which definition of "small" I wonder?

Offset(A0)	Size	Description
-$010	Long	Length of this block.
-$0C	Long	Either a pointer to the address of the I/O driver code which will free this block; or a pointer to the next free area in the common heap; or zero.
-$08	Long	ID of the job which owns this area of common heap.
-$04	Long	Address of a byte to be set when this area of heap is freed.

Table 7.1: Common Heap Header

Most jobs I've looked at, including many[3] of my own, don't bother with *user* heaps – see next section – and simply allocate space in the *common* heap as and when required. Perhaps the authors, myself included, need to think about which heap is best for the application?

7.1.1 Traps

Note that when allocating common heap space, the amount of RAM allocated will be rounded up to a multiple of 16 (or 32) bytes.

There are two traps to manage allocations in the common heap.

Sms.achp

This trap call used to be known as `MT_ALLOC` in QDOS, but is now called `SMS.ACHP` in SMSQ/E. The obligatory table of parameters can be seen in Table 7.2.

`D3` is interesting, Pennel doesn't mention it, but the SMSQ/E Manual (version 4.5) says that memory will be allocated in Fast RAM if `D3` is zero, and in ST compatible RAM if "acsi", on those machines with ST and Fast RAM.

Example of Use

Listing 7.1 shows a small example of requesting 1 byte of space from the common heap, then returning it to the heap. I tested the code by running it through QMON2 to check on the numbers returned in the appropriate registers and confirm the rounding and so on. I also checked the 16 bytes prior to the base address in `A0` and the overhead data can be seen there.

The figures listed are examples from my execution, your figures, should you attempt this code, will most likely differ,

```
sms.achp  equ $18                   ; Allocate common heap
sms.rchp  equ $19                   ; Free common heap

start
        moveq #sms.achp,d0          ; Trap code
        moveq #1,d1                 ; A single byte
        moveq #-1,d2                ; For this job
        moveq #0,d3                 ; For SMSQ/E on ST machines
        trap #1                     ; allocate
        tst.l d0                    ; Ok?
        bne.s allocated             ; Yes, free the heap space
        rts                         ; No, return error of S*BASIC
```

[3] Ahem, all!

Calling Parameters	
Register	**Usage**
D0.L	SMS.ACHP = $18
D1.L	Number of bytes required.
D2.L	ID of owning job. -1 indicates the current job.
D3.L	Zero, or “acsi” for Atari TT and machines with ST and Fast RAM.
Return Parameters	
Register	**Usage**
D0.L	Error code, or zero for no errors.
D1.L	Actual number of bytes allocated, including the 16 byte overhead.
D2	Corrupted.
D3	Corrupted.
A0.L	Base address of the allocated space. This is the first byte after the header for the block.
A1	Corrupted.
A2	Corrupted.
A3	Corrupted.
Errors in D0	
ERR.IJOB	Invalid job ID
ERR.IMEM	Out of memory

Table 7.2: SMS.ACHP Parameters

```
allocated
    moveq #sms.rchp,d0          ; Trap code
    trap #1                     ; Free heap space
    rts                         ; There are no errors, ever!
```

Listing 7.1: Common Heap Allocation Example

When I executed the code in Listing 7.1 and traced it with QMON2, I extracted the following detail:

- The base of the allocated block of RAM was at address $1014A0, this was the address passed back in `A0.L`;
- `D1.L` returned the value $20 – for 32 bytes allocated in total;
- The 16 bytes prior to the base address were:
  - `-16(A0)` = $00000020 = the size of the block allocated;
  - `-12(A0)` = $00000000 = Pointer to next free section of common heap, or driver deallocation code;
  - `-8(A0)` = $00000000 = Owning job Id;
  - `-4(A0)` = $00000000 = Address of byte to be set when this block is freed.

Interesting would you say? I requested a single byte yet I see that 32 bytes were allocated. This confirms that the 16 byte overhead is *included* in the allocated space value returned in `D1`, which none of the docs[4] mention.

I did some other tests with different request sizes and in all cases, it appears that the rounding is to

[4] Pennel, Dickens and QDOS/SMSQ Reference Manual, 4.5.

16 bytes and not to 8[5] as indicated in the documentation.

Sms.rchp

This trap call used to be known as `MT_RECHP` in QDOS, but is now called `SMS.RCHP` in SMSQ/E. The obligatory table of parameters can be seen in Table 7.3.

Calling Parameters	
Register	**Usage**
D0.L	SMS.RCHP = $19
A0.L	Base address of the allocated space.
Return Parameters	
Register	**Usage**
D0.L	Ignore, no errors are returned.
D1L	Corrupted.
D2	Corrupted.
D3	Corrupted.
A0L	Corrupted.
A1	Corrupted.
A2	Corrupted.
A3	Corrupted.
Errors in D0	
None – the trap call never fails.	

Table 7.3: SMS.RCHP Parameters

Listing 7.1 shows an example of the use of this trap.

7.1.2 Vectors

There are a pair of vectored routines, `MEM.ACHP` and `MEM.RCHP`, which enable code to manipulate space in the common heap. These vectors are atomic and must be called with the processor running in Supervisor Mode. They are normally used by device drivers to allocate space in the common heap for channel definition blocks for the Open function of the driver. The entire area allocated is zero filled if enough RAM existed in one contiguous block.

The 16 byte header for the area allocated is not filled in by the vectored code, it is the responsibility of the device driver code to do this.

Other than to mention that they exist, their use from within a device driver is beyond the scope of this eMagazine, and so they will not be discussed further.

7.2 User Heaps

If your job requires allocating small chunks of RAM, perhaps for a linked list, or a tree of structures, then rather than slicing and dicing the common heap into tiny bits, it is advisable to allocate a large, single, chunk of common heap and use that as a user heap to allocate the small chunks. The advantage of this process is that when done, you simple deallocate the common heap space and free up a large chunk on one go, rather than having to free up lots of small chunks. This helps to prevent fragmentation of the common heap.

[5] I wonder if Pennel says 8 because that's what QDOS did, but SMSQ/E uses a 16 byte rounding instead?

The steps involved in this process are:

- Allocate a suitable sized area of common heap to be used for your user heap;
- Link the allocated area into your job's user heap space;
- Allocate space in the user heap, as and when required;
- Use and abuse the user heap space allocated;
- Optionally, but good practice, deallocate used space when finished with;
- Release the common heap at job end – this may be manually done, or left to QDOS/SMSQ to do it automatically.

Note that when allocating user heap space, the amount of RAM allocated will be rounded up to a multiple of 8 bytes. Each chunk of user heap allocated will have an 8 byte overhead on top of the space requested, and possibly rounded up. For example:

- Request 10 bytes of user heap;
- The allocation will be 16 bytes after rounding up;
- The total space allocated from the user heap's free space will be 16 as the 8 byte overhead is *not added* to the requested size!

The 8 byte overhead holds the data detailed in Table 7.4.

Offset	**Size**	**Description**
$00	Long	Length of this block.
$04	Long	A *relative* pointer to the next free space in the user heap space.

Table 7.4: User Heap Header

The overhead will be written *to the start of the allocated heap space*, at the address returned in `A0`, and is considered part of the user heap allocation. This is different to the overhead in common heap space.

You will need to save the size of each and every user heap allocation so that it can be returned to the heap's free space when done with. You can do this manually – possibly dangerous if you get the sizes wrong – if you only have a couple of allocations, or request 8 bytes more for each allocation and use the space allocated from address `8(A0,A6)` on return, rather than that from `0(A0,A6)`. You can also, if applicable, simple rerun the code to create the user heap space which will free up all allocated chunks in the user heap in one fell swoop.

7.2.1 Traps

Note that when allocating user heap space, the amount of RAM allocated will be rounded up to a multiple of 8 bytes. Each allocation will itself have an 8 byte overhead as discussed in Table 7.4, the 8 bytes is taken out of the space requested and is the first two long words in the allocated section of the heap.

In other words, if you need 10 bytes, ask for 18 because the first 8 bytes, at the address returned in `A0`, will contain the 8 byte overhead. This is useful to keep a hold of as the length of the block is needed when returning the allocated RAM back to the user heap free space with `SMS.REHP`.

All addresses are relative to `A6` when allocating or deallocating user heap space.

Sms.alhp

Table 7.5 shows the registers that need setting up to call the SMS.ALHP trap to allocate memory in a user heap. The return parameters are interesting too.

Calling Parameters	
Register	**Usage**
D0.L	SMS.ALHP = $0C
D1.L	Number of bytes required. Does not include the 8 byte overhead. Perhaps ask for 8 extra bytes?
A0.L	Pointer to a pointer to the free space list. Relative to A6.
A6.L	Base address of job.
Return Parameters	
Register	**Usage**
D0.L	Error code, or zero for no errors.
D1.L	Actual number of bytes allocated – should be the number requested and includes the 8 byte overhead.
D2	Corrupted.
D3	Corrupted.
A0.L	Base address of the allocated space. This points to the first byte of the header for the block. Relative to A6.
A1	Corrupted.
A2	Corrupted.
A3	Corrupted.
A6	Preserved.
Errors in D0	
ERR.IMEM	No area of free space was large enough to allocate.

Table 7.5: SMS.ALHP Parameters

Sms.rehp

This trap call is used when initially adding an area of RAM to be used as a user heap, or when freeing an allocation within the user heap. In the former case, the long word at (A6,A1.L) should be zero, in the latter, it will be some other, non-zero value. Listings 7.2 and 7.4 show examples of both.

User Heap Example

Listing 7.2 shows a small example where a chunk of 64 Kb of common heap is requested from the system, and, if successfully allocated, is converted to a user heap. the code checks return values from traps – it's assembled to be CALLed from SuperBASIC/SBASIC – and if any errors occur, the code exits back to SuperBASIC/SBASIC with the error code.

The allocated area of common heap is then converted to a user heap by simply making sure that the free space address, pointed to by the long word at myHeap, is zero, then calling the SMS.REHP trap.

```
sms.achp   equ  $18
```

Calling Parameters	
Register	**Usage**
D0.L	SMS.REHP = $0D
D1.L	Length of space to link (back) into a heap
A0.L	Base address of the space to be linked in/back. Relative to A6.
A1.L	Pointer to a pointer to the free space list. Relative to A6.
A6.L	Base address of job.
Return Parameters	
Register	**Usage**
D0.L	Ignore, no errors are returned.
D1L	Corrupted.
D2	Corrupted.
D3	Corrupted.
A0L	Corrupted.
A1	Corrupted.
A2	Corrupted.
A3	Corrupted.
Errors in D0	
None – the trap call never fails.	

Table 7.6: SMS.RCHP Parameters

```
sms.rehp  equ $0d

; First, allocate a 64Kb common heap area:
start
    moveq #sms.achp,d0              ; Trap code
    move.l #65536,d1                ; 64Kb required
    moveq #-1,d2                    ; This job will be the owner
    trap #1                         ; Allocate the space
    tst.l d0                        ; Did it work?
    beq.s heapOk                    ; Yes

; Handle out of memory errors here.
    rts                             ; Back to SuperBasic

; We have a common heap, convert it to a user heap :
heapOk
    moveq #sms.rehp,d0              ; Trap code
    suba.l a6,a0                    ; We need A0 to be A6 relative
    lea myHeap,a1                   ; Pointer to heap header
    move.l 0,(a1)                   ; Indicate this is a new heap
    suba.l a6,a1                    ; This needs to be relative A6
    trap #1                         ; That should do it (No errors)
    bra.s useHeap                   ; Go and use the heap space

myHeap
    ds.l 1                          ; Pointer to free space
```

Listing 7.2: User Heap Creation Example

Once we have an area of RAM set aside as a user heap, we can begin to use it. Listing 7.3 is an example of allocating 200 bytes from the newly created user heap.

```
sms.alhp  equ $0c

; Allocate space in the user heap.
useHeap
    moveq #sms.alhp,d0          ; Trap code
    move.l #200,d1              ; I need 200 bytes of user heap
    lea myHeap,a0               ; MyHeap = Free space pointer
    suba.l a6,a0                ; Relative to A6
    trap #1                     ; Allocate 200 bytes
    tst.l d0                    ; Did it work?
    bra.s doStuff               ; Go and use the allocation

; Handle out of user heap memory errors here
    rts                         ; Back to SuperBasic

; Now we have allocated some user heap, use it somehow
doStuff
    adda.l a6,a0                ; Absolute the address
```

Listing 7.3: User Heap Allocation

At this point, running via QMON2, I checked the allocated user heap space to see if the 8 byte overhead was prior to the address returned in `A0`, as per the common heap; or at the returned address. The overhead is indeed at the address pointed to by `(A6,A0.L)` on return and after "unrelativing" the address, `0(A0)` holds the long word $000000C8 which is the length of the block, the long word at `4(A0)` is zero.

When we have finished using the 200 bytes, we can return it to the user heap, in case we need more space at some other point in the code. This uses the same trap call which created the user heap in the first place, but this time, the pointer to the free space, `myHeap`, will not be zero as it points to the first free chunk of user heap.

```
freeUser
    moveq #sms.rehp,d0          ; Trap code
    move.l #200,d1              ; Size is 200 bytes
    suba.l a6,a0                ; A0 has to be relative a6
    lea myHeap,a1               ; Pointer to top of heap
    suba.l a6,a1                ; Which has also to be relative A6
    trap #1                     ; Deallocate the 200 byte area
    rts                         ; Back to SuperBasic
```

Listing 7.4: User Heap Deallocation

You will note that I have hard coded the block size in register `D1` for this trap call. This is one way to do it especially if you only required a couple of chunks of user heap, keeping a note of the sizes isn't difficult in that case. However, if you are allocating lots of user heap space, or chunks of many different sizes, what to do?

My advice would be, allocate space for 8 bytes plus what your code needs, and use the data from `8(A6,A0.L)` onwards, and do not touch anything below that address. When you are done with the space and about to free it, simply load `D1` from `0(A6,A0.L)` to get the block size, and *Robert is your mother's brother*. Obviously, if you have "unrelatived" the base address of the allocated space, you would use the data from `8(A0)` and load `D1` from `(A0)` prior to freeing the space again.

Once all the user heap space has been freed up and is no longer required, the chances are that your application is about to exit. At this point, it could exit and automatically free up the 64 Kb chunk of common heap, without any further work on the code's part, or, the code could be nice and free it's own allocation with the SMS.RCHP trap call.

Relative Addresses

In the listings above, you will note that I add or subtract A6 from A0 and A1 at various places in the code. This is because when using these traps to manipulate user heap space, those registers have to be relative to A6. It's a bit of a faff and there are a couple of ways around this problem:

- Do as I have done, and add or subtract A6 as necessary, then address user heap areas using offsets on (A0) as required.
- Do all your addressing as offsets on (A6,A0.L) as necessary, although this addressing mode takes 2 extra clock cycles over just (A0)[6];
- Zero A6 at the start of the code, and then the addresses will be both absolute and relative at the same time, so you can use offsets on (A0) or (A6,A0.L) as you prefer;
- Use the appropriate vector calls rather than the trap calls, those use absolute addresses. Speaking of which....

7.2.2 Vectors

There are a pair of vectored routines, MEM.ALHP and MEM.REHP, which are *non-atomic*[7] versions of the user heap trap calls. They take exactly the same parameters as the two trap calls, but do not require A6 to be considered. Even better, there is no need to mess around keeping everything relative to A6 as the two vectors don't care about such necessities!

Converting the listings above to use vectors instead of traps, gives us the code in Listing 7.5.

```
sms.achp   equ  $18
mem.rehp   equ  $DA

; First, allocate a 64Kb common heap area:
start
    moveq #sms.achp,d0              ; Trap code
    move.l #65536,d1                ; 64Kb required
    moveq #-1,d2                    ; This job will be the owner
    trap #1                         ; Allocate the space
    tst.l d0                        ; Did it work?
    beq.s heapOk                    ; Yes

; Handle out of memory errors here.
    rts                             ; Back to SuperBasic

; We have a common heap, convert it to a user heap :
heapOk
  move.w mem.rehp,a2              ; Vector
    lea myHeap,a1                   ; Pointer to pointer to free space
    clr.l (a1)                      ; Initialise user heap
    move.l #65536,d1                ; We have 64 Kb to play with
    jsr (a2)                        ; Convert to a user heap
    tst.l d0                        ; Did it work
```

[6] True, but each and every ADD or SUB of A6 to/from A0 or A1 will cost you 6 clock cycles, so there!

[7] In other words, the operation could get interrupted and the scheduler entered.

```
    beq.s  useHeap              ; Yes
    rts                         ; No

myHeap
  ds.l  1                     ; Pointer to free space

mem.alhp   equ $D8

; Allocate space in the user heap.
useHeap
    move.l  #200+8,d1           ; I need 200 bytes of user heap
    lea  myHeap,a0              ; MyHeap = Free space pointer
    move.w mem.alhp,a2          ; Vector
    jsr  (a2)                   ; Get some user heap
    tst.l  d0                   ; Did it work?
    bra.s  doStuff              ; Yes
    rts                         ; No

; Now we have allocated some user heap, use it somehow
doStuff
   move.l #$12345678,8(a0)      ; Avoid writing the header bytes
   ...

; Now deallocate the user heap space.
freeUser
    lea  myHeap,a1              ; Pointer to pointer to free space
  move.w mem.rehp,a2          ; Vector
    move.l  (a0),d1             ; Block length to free up
    jsr  (a2)                   ; Free the space
    rts                         ; Back to SuperBASIC
```

Listing 7.5: User Heap Vectors Example

If you trace the code using QMON2[8] then you will see that when you arrive at the label `doStuff`, the base address of the user heap allocated at `(A0)`, holds the length of the block, which is 208 or $D0[9].

Freeing User Heap Space Quickly

If you have, for example, some deeply recursive code which allocates space in the user heap, how do you cope with an error whereby you have to free up all the allocated bits that were ok until the problem arrived? I'm thinking perhaps of an expression evaluator as a specific example, but it could be a parser or a compiler building a symbol table or parse tree etc. I'm also thinking of the problem where the code doesn't just give up, but informs the user – *expression too complex* or *invalid operation* etc – but then loops back to the prompt for more input.

If, in the case of the expression evaluator, the code will have allocated lots of chunks of user heap to build the expression tree[10] then those nodes in the tree need to be deallocated before the next expression can be evaluated, otherwise, at some point, the user heap space will be full of nodes that are no longer required, but are hogging all the space.

[8] Other monitors are available....

[9] And that value confused me as QMON listed the instruction to load `D1` as `MOVE.L #$D0,D1` and I was initially confused as I didn't have an instruction to move `D0` into `D1`. Then I read the screen a little bit better and understood!

[10] Usually an Abstract Syntax Tree or AST.

The easiest manner of deallocating all allocated space in the user heap is simply to clear the pointer to the free space to zero, then call the SMS.REHP trap or the MEM.REHP vector and link the entire user heap to free space again. Something similar to Listing 7.6.

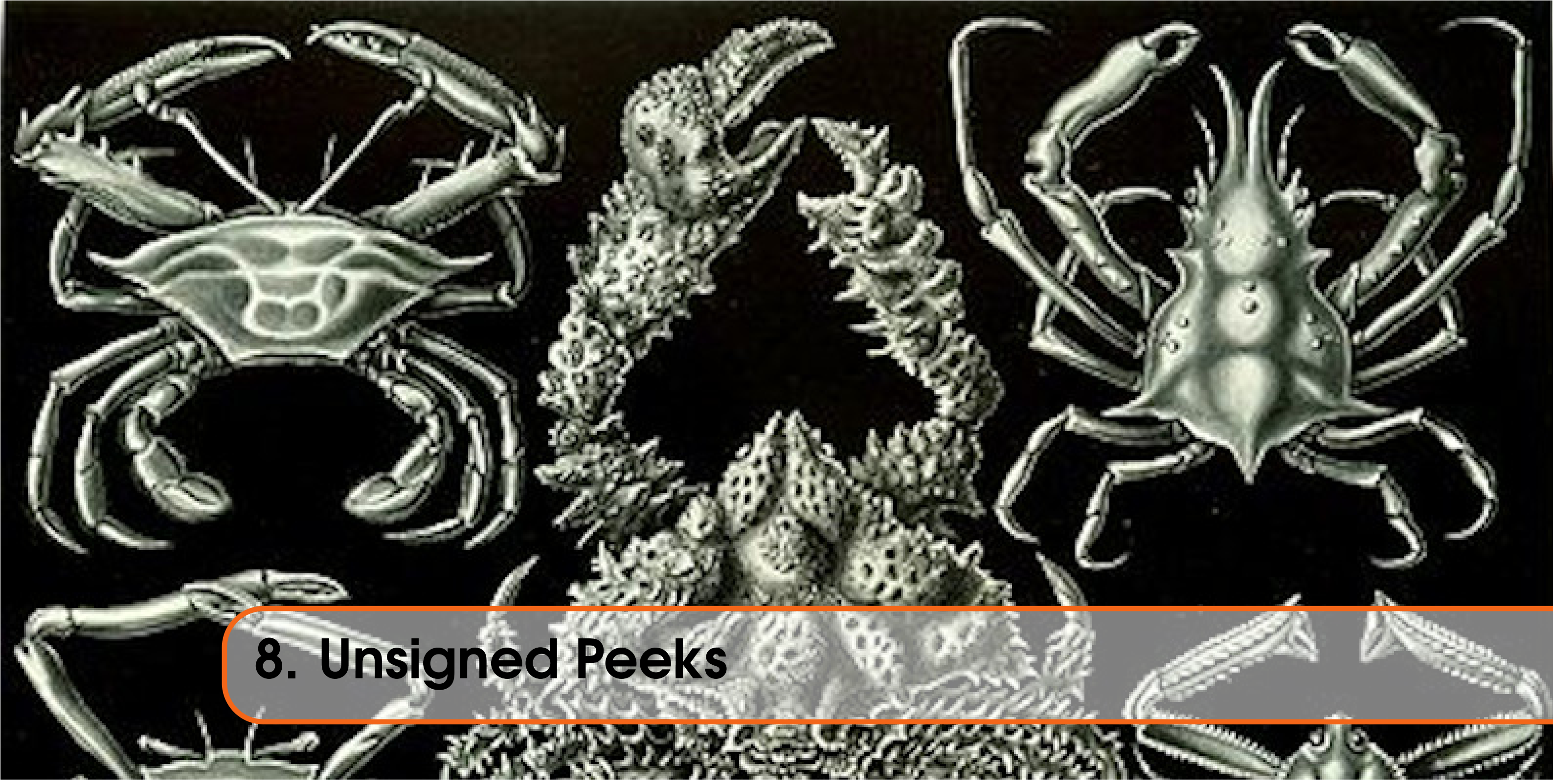
```
start
    bsr getCommonHeap          ; Allocate heap space or die
    bra.s mainLoop             ; Skip to main loop

userHeap
    dc.l 1

; On first entry:
; Link the allocated space in the common heap into a user heap.
;
; On subsequent entries:
; Wipe everything from the user heap.
mainLoop
    lea myHeap,a1              ; Pointer to pointer to free space
    clr.l (a1)                 ; Initialise user heap
    move.w mem.rehp,a2         ; Vector
    move.l #heapSize,d1        ; Size of user heap
    jsr (a2)                   ; Convert/wipe user heap
    tst.l d0                   ; Did it work
    beq.s useHeap              ; Yes
    rts                        ; No, exit with error code
;
; None of these will return if an error occurs. The errors
; will be handled below, the stack unwound and any user
; heap allocations freed.
    bsr getUserInput           ; Get next expression or exit
    bsr lexer                  ; Build token list
    bsr parser                 ; Build AST
    bsr evaluate               ; Evaluate the expression
    bra.s mainLoop             ; Keep going

lexError
    bsr doLexError             ; Handle lexer errors
    bra.s mainLoop             ; And go again

parseError
    bsr doParseError           ; Handle parser errors
    bra.s mainLoop             ; And go again

EvalError
    bsr overflow               ; Check/Handle overflow
    bsr divZero                ; Check/handle divide by zero
    bsr ...                    ; Etc
    bra.s mainLoop             ; And go again

    ...
```

Listing 7.6: User Heap Total Deallocation

On the first entry to mainLoop, the common heap allocation is linked into the user heap's free space, the whole allocation is free for use as a user heap. On subsequent passes through mainLoop, the user heap is effectively reinitialised, thus freeing up every piece of allocated space which was allocated before the code went into error recovery.

Not shown in the example code above is the handling of the `A7` stack, which needs to be preserved at the start of the `mainLoop` and reset after each and every error so that it is correctly set each time we pass by the `mainLoop` address.

Can you tell I'm writing an expression evaluator then?



8. Unsigned Peeks

There's a thread on the QL Forum[1] about how difficult it might be to write programs in S*BASIC on SMSQ/E. There is a pile of useful information there. One of the more interesting comments was about a hardware timer to get proper delays in code, but this required the use of PEEK_L. And, as we all know, PEEK_L returns a float, and is signed.

I responded on this post[2] with an example of some code to get around that problem by adding 2^{31} if the result was negative. Sadly, in my posting, I actually typed 2^{32}– Duh!

Marcel responded that *someone could do a PEEK_UL function* to get an unsigned result. This caused confusion for a bit as nobody knew that PEEK_UL existed and when they tried, it was, of course, not found. Marcel *may* have been hinting that *someone could easily write an unsigned version of the PEEK_L (and PEEK_W) functions*. So I did! And here they is.

You'd *think* there was nothing to it really, and there isn't, but along the way to producing this code, I had to delve into some old code of mine[3] to convert a long value to a floating point value on the maths stack. For the life of me, I couldn't remember how I did it, so I fell down that rabbit hole until I had a vague understanding of QL Floating Point and the conversion code.

SMSQ/E has a maths package operation to convert a long value on the maths stack into a float, however, QDOS doesn't have this option – it's interestingly missing from the maths package op code – so there are two versions of the source code and binary files on the accompanying code download for this issue.

The SMSQ/E version is 230 bytes while the QDOS version is longer, at 258. The QDOS version is, however, safe to run on SMSQ/E as well. Just don't try running the SMSQ/E version on QDOS – I have no idea what will happen.

[1] https://qlforum.co.uk/viewtopic.php?f=3&t=4027&sid=7a218630d06b1d78c6bb436e1fbf32f9

[2] https://qlforum.co.uk/viewtopic.php?f=3&t=4027&p=45290#p45290

[3] Well, it's either mine or I stole it from somewhere, I can't remember that far back!

8.1 PEEK_UW and PEEK_UL

PEEK_W and PEEK_L on the QL are signed. They return positive and negative values. This is sometimes a bit of a pain because, taking PEEK_L as an example, fetching any value over $7FFFFFFF or 2,147,483,647, will result in the value going negative. This may not be what you desire. The solution is easy, add 2^{15} or 2^{31}, depending on whether you are peeking for words or longs, to the result on return to S*BASIC. Sometimes you forget, well, I do, and so, these two new functions will do the testing and adding for us, leaving S*BASIC to get on with interpreting the rest of the code.

Listing 8.1 is the start of the source code, and shows the various equates I've used in the code. These are, as mentioned in the News Section, now using the SMSQ/E mnemonics, don't worry if you are using QDOS as it's the values that count – I could have called them Fred, Barney Wilma, Betty etc, but that's not a meaningful set of names[4].

```
sb_arthp equ $58                    ; Where is top of maths stack
sb.inipr equ $110                   ; Add procs/fns to S*BASIC
sb.gtlin equ $118                   ; Fetch long int parameters
qa.resri equ $11a                   ; Reserve maths stack space
qa.op    equ $11c                   ; Do one maths stack operation

qa.flong equ $09                    ; Convert long to float (SMSQ)
qa.add   equ $0a                    ; Add TOS to NOS on maths stack

err.ipar equ -15                    ; Bad parameter error code

exp231   equ $0820                  ; 2^31 exponent
exp215   equ $0810                  ; 2^15 exponent
mantBoth equ $40000000              ; 2^31 & 2^15 mantissa

peek_w   equ exp215                 ; Flag for peek_uw
peek_l   equ exp231                 ; Flag for peek_ul
```

Listing 8.1: Unsigned Peeks - Equates

The only thing I'd draw your attention to is the two flags I've set up to determine if we are peeking words or longs, those being peek_w and peek_l. I've given those a value which just happens to correspond to the exponent for a floating point value of either 2^{15} or 2^{31} – this is a cunning plan to save me some work later on!

The code proper begins at the label start, which you can see in Listing 8.2 where we have the standard code to link in new procedures and/or functions, and the requisite definition block in Listing 8.3 where we are defining no new procedures, only two new functions, PEEK_UL and PEEK_UW.

```
start
    lea define,a1                   ; Proc/FN definition list
    move.w sb.inipr,a2              ; Ready to add to S*BASIC
    jsr (a2)                        ; Do it
    rts                             ; Take errors back to S*BASIC
```

Listing 8.2: Unsigned Peeks - Linking into S*BASIC

```
define
    dc.w 0                          ; No procedures
```

[4] Unless we are writing a Flintstones application I suppose.

```
    dc.w 0                          ; End of procedure list

    dc.w 2                          ; 2 Functions
        dc.w peek_ul-*              ; Offset to function
        dc.b 7,'PEEK_UL'            ; The function name

        dc.w peek_uw-*              ; Offset to function
        dc.b 7,'PEEK_UW'            ; The function name
    dc.w 0                          ; End of functions
```

Listing 8.3: Unsigned Peeks - Procedure/Function definition block

Both new functions share a lot of common code. To this end, I set a flag to determine which is being called so that the places where things differ, can easily tell the functions apart. PEEK_W begins, as Listing 8.4 shows, by setting the correct flag in D4.W, before jumping off to join the first section of common code.

```
peek_uw
    move.w #peek_w,d4               ; Flag for peek_uw
    bra.s peekBoth                  ; Off we go then!
```

Listing 8.4: Unsigned Peeks - Peek_uw

Listing 8.5 is the beginning of the PEEK_UL function. It also sets a flag before dropping into the common code for both functions.

```
peek_ul
    move.w #peek_l,d4               ; Flag for peek_ul

peekBoth
    move.l a5,d0                    ; First parameter
    sub.l a3,d0                     ; Last parameter
    cmpi.l #8,d0                    ; 8 bytes per parameter
    beq.s pulGetParam               ; We have one parameter to get

pulBadParam
    moveq #err.ipar,d0              ; That didn't go well then!
    rts                             ; Back to S*BASIC
```

Listing 8.5: Unsigned Peeks - Peek_ul

Both functions require a single parameter, so we subtract A3.L from A5.L and if the result is not 8 bytes, then we exit back to S*BASIC with a bad parameter error. A3.L points to the last function parameter on the name table, and A5.L to the first. Each name table entry takes 8 bytes.

If we do have 8 bytes of a difference, then we know that there's exactly one parameter waiting, so we can go fetch it. Listing 8.6 shows the code to fetch the address parameter as a long value.

```
pulGetParam
    movea.w sb.gtlin,a2             ; We want a long integer
    jsr (a2)                        ; Go fetch
    beq.s pulTestOne                ; No problems detected
    rts                             ; Back to S*BASIC with error
```

Listing 8.6: Unsigned Peeks - Fetching one parameter

When fetching parameters for a procedure or function, D3.W holds the count of parameters actually fetched. Just as an extra check, we test this, in Listing 8.7, to make sure that we did indeed only fetch a single parameter.

```
pulTestOne
    cmpi.w #1,d3                ; Make sure we got one only
    bne.s pulBadParam           ; Weirdness has happened!
```

Listing 8.7: Unsigned Peeks - Testing parameters fetched

If we somehow managed to fetch more, or less, than one parameter, we bale out to S*BASIC with a bad parameter error. If not, then we continue to pull the passed address off of the maths stack into `A2.L` and then we 'peek' the long value at the requested address into `D7.L` as per Listing 8.8

```
pulGotParam
    move.l (a6,a1.l),a2         ; Get the address into a2.
    move.l (a2),d7              ; Peek the address
```

Listing 8.8: Unsigned Peeks - Getting the address, and peeking it

At t his point, we have pretty much executed a `PEEK_L` function call, but we might want only the first word if we are executing `PEEK_UW`. If so, we need to return only the current high word of `D7.L`, so Listing ? is the code that does the check. If we are indeed running `PEEK_UW`, then the low word is cleared and then the upper word is swapped into the lower word, giving the correct value in `D7.L` for `PEEK_UW`.

```
pulWord
    cmpi.w #peek_w,d4           ; Are we looking for a word?
    bne.s pulGotPeek            ; No, continue
    clr.w d7                    ; We don't need the bottom word
    swap d7                     ; Correct word for peek_uw
pulGotPeek equ *                ; It's just a label for both
```

Listing 8.9: Unsigned Peeks - Fixup for PEEK_UW

Both functions come together again at label `pulGotPeek`, where `D7.L` holds the possibly negative value we need to return to S*BASIC. Unfortunately, this is also exactly where they diverge again, however, I'm only discussing the SMSQ/E version here. I've explained converting a long word into floating point the end of this chapter, if you are interested?

The differences are these:

- SMSQ/E has a maths package operation, `OP.FLONG`, to take the long word on the top of the maths stack, and convert it to a floating point value at TOS, adjusting `A1.L` as necessary (subtracting 2 from it – the difference between a 6 byte float and a 4 byte long)
- QDOS requires the developer, me, to allocate an additional 2 bytes on the maths stack. There is already 4 bytes available as we pulled the address parameter as a long word, which takes 4 bytes.
- It the space allocation succeeds then we can convert, manually, and normalise the long word in `D7.L` into a float and stack it.
- In the QDOS variant, the developer, me again, has to be very careful to keep the TOS pointer in A1 correct at all times.

Listing 8.10 is the simple way that SMSQ/E converts a long word to a floating point value.

```
pulFloatD7SMSQ
    move.l d7,(a6,a1.l)         ; Stack D7.L
    move.l #qa.flong,d0         ; Float a long operation code
    bsr.s pulDoMathsOp          ; Do it
    beq.s pulTestNegative       ; All was well
```

```
        rts                             ; Take errors back to S*BASIC
```

Listing 8.10: Unsigned Peeks - Floating a Long SMSQ/E style

We need to stack the value in D7.L so we can do that easily as we know that there are 4 bytes available on the maths stack, and that A1.L is still correctly pointing at the TOS where we need our long word to be. As with almost everything S*BASIC, addresses are relative to A6.

After stacking D7.L, we simply call a subroutine, pulDoMathsOp, which can be seen later in Listing 8.15, to do the hard work of converting the value. If that worked, then we skip off to pulTestNegative, where we join up with the code for QDOS again.

```
pulTestNegative
        btst  #7,2(a6,a1.l)             ; Mantissa bit 31
        beq.s pulValuePositive          ; It's positive, skip
```

Listing 8.11: Unsigned Peeks - Testing for negativity

Listing 8.11 checks the byte at the high end of the mantissa for the floating point value on the maths stack. Bit 31 of the mantissa is the sign bit and that corresponds to bit 7 in this particular byte. If the bit is clear, then we have a positive number as there's no need to do the addition to make it positive.

If the bit is set, then the value is negative, and we need to add 2^{15} or 2^{31} depending on the function currently executing. Listing 8.12 shows how I decided to do it.

```
pulValueNegative
        moveq  #6,d1                    ; Need space for a float
        bsr.s  pulGetSpace              ; Reserve space, sets A1
        move.w d4,(a6,a1.l)             ; Stack 2^D4 exponent
        move.l #mantBoth,2(a6,a1.l)     ; Stack common mantissa
        move.l #qa.add,d0               ; Add TOS to NOS operation code
        bsr.s  pulDoMathsOp             ; Do it
        beq.s  pulValuePositive         ; All was well
        rts                             ; Take errors back to S*BASIC
```

Listing 8.12: Unsigned Peeks - Adjusting for negativity

The first thing we have to do is request 6 bytes, enough room for a new floating point value, on the maths stack. This is done by a subroutine, pulGetSpace in Listing 8.14 which will make sure that the space is allocated and adjusts A1.L as necessary.

We can now stack the floating point representation of 2^{15} or 2^{31} appropriately. Remember that flag I used to determine whether we were in PEEK_UW or PEEK_UL? Well, that value just happens to be the exponent for 2^{15} or $2^{31}$and so we can stack the exponent directly from the flag register, D4.W. The mantissa for both values is the same, so that gets stacked next. We now have (A6,A1.L) pointing at 2^{15} or 2^{31} on TOS, and the peek value at Next On Stack, NOS. We need to add them together.

SMSQ/E and QDOS both have the same maths package operation to add two floats, QA.ADD. This has the side effect of replacing NOS with the result of the addition and removes TOS making the existing NOS the new TOS. A1.L is now 6 larger than previously, and we need this to be saved away. The subroutine pulDoMathsOp, in Listing 8.15 does this for us.

The value at TOS is now guaranteed to be positive, so we can carry on and return the result to S*BASIC. Listing 8.13 is how we do that.

```
pulValuePositive
        moveq  #2,d4                    ; Return is floating point
        moveq  #0,d0                    ; No errors
```

```
    rts                               ; Done
```

Listing 8.13: Unsigned Peeks - Return to S*BASIC

D4.W is set to 2 to indicate a float result is on the maths stack at TOS and we return with no errors.

That's basically all there is to it! Except for a couple of small subroutines. Listing 8.14 is the subroutine pulGetSpace which requests space for the maths stack and keeps the various pointers correctly aligned.

```
pulGetSpace
    move.l d1,-(a7)                   ; Save space required
    move.l a1,sb_arthp(a6)            ; Store current TOS
    move.w qa.resri,a2                ; Allocate maths stack space
    jsr (a2)                          ; Do it - never errors
    move.l sb_arthp(a6),a1            ; Fetch possible new A1 value
    move.l (a7)+,d1                   ; Retrieve space requested
    sub.l d1,a1                       ; Adjust space as required
    move.l a1,sb_arthp(a6)            ; And store new TOS
    rts
```

Listing 8.14: Unsigned Peeks - Allocating maths stack space

The code expects D1.L to contain the number of bytes needed and A1.L to point at the maths stack TOS, relative to A6.L. The QA.RESRI vector code trashes registers D2, D3, A1 and A2, but we have to reload A1.L from SB_ARTHP anyway, so that's no matter.

The subroutine requests D1.L bytes of space, then retrieves the potentially new value for A1.L from SB_ARTHP before subtracting D1.L to get the new TOS. This value is saved back in SB_ARTHP as it must be.

On exit, A1.L is the new TOS pointer, ready for use by our code.

Listing 8.15 is the final subroutine. This one, pulDoMathsOp, executes a single maths package operation according to the code in D0.W.

```
pulDoMathsOp
    move.w qa.op,a2                   ; Do 1 maths operation
    jsr (a2)                          ; Do the conversion
    beq.s pulFixup                    ; No errors, adjust SB_ARTHP
    rts                               ; Errors go back to caller

pulFixup
    move.l a1,sb_arthp(a6)            ; Save new TOS pointer
    moveq #0,d0                       ; No errors
    rts
```

Listing 8.15: Unsigned Peeks - Execute maths package operation

The subroutine expects an operation code, as a word, in D0.W and executes it. If all went well, then the TOS might have changed, so to keep things right, the potential new TOS is pulled from SB_ARTHP back into A1.L before setting no errors in D0, and returning to the caller.

If errors were detected, the code makes an early exit back to the caller with D0 set to the error code, and the Z flag unset.

8.1.1 QDOS Conversion of Long to Float

AS mentioned above, QDOS requires manual intervention to convert a long word into a floating point value. We already have room for a long word, but we need an additional two bytes for a float, so first of all, we have to request two extra bytes. Listing ? shows how we do this by setting `D1.L` to 2, and calling the `pulGetSpace` subroutine, in Listing ?, to do the hard work of getting the space and keeping the pointers aligned. On return, `A1.L` is set ready for us to use to stack a 6 byte float value.

```
pulMakeSpace
    moveq #2,d1                   ; 2 extra bytes required
    bsr.s pulGetSpace             ; Reserve space, sets A1 to TOS
```

Listing 8.16: Unsigned Peeks - QDOS Making stack space

Unfortunately, QDOS is bereft in the maths package operations to convert a long word to a float, so we have to do it manually. Listing 8.17 does the hard work for us. At the end, we should have the exponent value in register `D5.W` and the mantissa in register `D6.L`.

```
pulFloatD7QDOS
    move.w d7,d5                  ; Exponent
    move.l d7,d6                  ; Mantissa
    beq.s pulNormalised           ; We are done if D7.L = zero
    move.w #$081f,d5              ; Starting exponent value
    add.l d7,d7                   ; Already normalised?
    bvs.s pulNormalised           ; Yes
    subq.w #1,d5                  ; No, halve the exponent
    move.l d7,d6                  ; Mantissa * 2
    moveq #$10,d0                 ; Start with a 16 bit shift

pulNormalise
    move.l d6,d7                   ; Copy the mantissa
    asl.l d0,d7                   ; Multiply by 2^D0
    bvs.s pulTooBig               ; Oops, too big now
    sub.w d0,d5                   ; Adjust exponent for the shift
    move.l d7,d6                  ; Getting more normalised now

pulTooBig
    asr.w #1,d0                   ; Halve the shift size
    bne.s pulNormalise            ; Do 8, 4, 2, 1 shifts

pulNormalised
    move.w d5,(a6,a1.l)           ; Stack exponent
    move.l d6,2(a6,a1.l)          ; Stack mantissa
```

Listing 8.17: Unsigned Peeks - QDOS Converting Long to Float

How does it work? I was hoping you wouldn't ask!

8.1.2 QL Floating Point Values

The QL's internal floating point format is a 2 byte exponent and a 4 byte mantissa. What are they? Well, if you have a number such as 1.23×10^3 then the mantissa is the 1.23 and the exponent is 3 and the number is "normalised" when the mantissa is "n.something". Zero is a special case.

1.23×10^3 could easily be written as 12.3×10^2 or 123×10^1 they all result in 1,230. A properly normalised number, in this notation, is a single digit prior to the decimal point, so 1.23×10^3 is the correct value.

The exponent is the power of 10 that the mantissa is to be multiplied by, to get the actual[5] value, in other words, the value is:

$$mantissa * 10^{exponent}$$

The same is true in binary but there the value is $mantissa * 2^{exponent}$ as we are working in base 2, not base 10.

In SMSQ/E, the mantissa is normalised when bit 31 or bit 30 of the mantissa is set. Bit 31 will be set if the number is negative, and bit 30 if positive. The exponent only has the bottom 12 bits available as the top nibble is used to determine if this is a decimal float value, a binary float value or a hexadecimal float value.

12 bits is 2,048. $800 in hexadecimal, so the exponent has 12 bits of precision. However, it is offset by $800, so the range is from $2^{-2,048}$ to $2^{2,047}$ which is from 3.094^{-617} to 1.616^{616} which is a decent enough range. Because the exponent is offset, this makes the floating point value equal to:

$$mantissa * 2^{exponent - \$800}$$

Zero is a special case where the exponent and mantissa are both zero.

The bits in the mantissa represent fractions, but in power's of 2. They can be thought of as $\frac{1}{2^{31-bit}}$ but , as with many things mathematical, this can also be expressed as $2^{-(31-bit)}$. The fractional parts of the mantissa start with bit 30, the most significant bit, which represents $\frac{1}{2^1}$, 2^{-1} or 0.5, down to bit 0, representing $\frac{1}{2^{31}}$, 2^{-31} or *not very much at all*![6]

A couple of examples:

- 10.0 is represented as $0804 50000000
- -10.0 is represented by $07FC D0000000

Taking 10 first.

The mantissa is %0101 0000 0000 0000 0000 0000 0000 0000 so we are only interested in the %0101 bits, the trailing zeros can be ignored.

The sign bit is zero, so the number is positive. So far so good. Bit 30, which represents 0.5 is set, as is bit 28 which represents 0.125, the mantissa is therefore the addition of these fractions. The result is 0.625.

The exponent is $0804, subtracting $0800 we get $04, so 10 is 0.625 multiplied by 2^4 or 16. 10 is indeed equal to 0.625 times 16.

Negative 10 next.

The mantissa is %1101 0000 0000 0000 0000 0000 0000 0000 and again, we can ignore all the trailing zero, keeping only %1101.

[5] Watch out for errors though, floating point is notorious for errors. Many floating point values cannot be represented exactly. One third for example.

[6] Ok, ok, it's $\frac{1}{2,147,483,648}$, which is pretty small.

Bit 31 is 1 so we are dealing with a negative number.

Bits 30, 0.5, and 28, 0.125, are both set, so we have a value again, of 0.625. Subtracting \$800 from \$7FC results in \$FFFC or -4, and 0.625 times 2^{-4} is -10.

If we assume `D7.L` is zero, then we have the easy case where we move it into `D5.W` and `D6.L`, it will set the `Z` flag and we are ready to stack the value.

If we take `D7` as having the value \$1E240 which is 123,456 in decimal. We start off by setting up the various registers, `D5` gets a copy of the low word, `D6` the whole \$1E240. Because we are not dealing with zero, we drop down to set `D5` to the first guess exponent, \$081F, and adding `D7` to itself which results in `D7` = \$3C480. This didn't overflow, so we still have a valid mantissa which is twice as big as it was.

As we have now doubled the mantissa, we must also halve the exponent, which means subtracting 1 from `D5` to get \$081E.

The new mantissa is copied into `D6` where we will keep it for those occasions when shifting it by `D0` bits results in overflow. `D0` is initialised to \$10 – we will attempt a 16 bit shift first of all.

We *could* shift by a single bit each time, and halve the exponent each time there's no overflow, but this method is quicker. If `D7` *can* be shifted 16 bits left, then that that's one shift, not 16[7].

We are now at label `pulNormalise` for the first time, and here we enter the normalisation loop. We will multiply `D7` by "lots of bits" each time as long as we don't get overflow. If we do hit overflow, we have gone too far and need to back up a bit, and multiply by half as many "lots of bits".

We copy the most recent valid mantissa from `D6` to `D7` ready to try again, and shift it left by the \$10 bits held in `D0`. This caused overflow as `D7` was holding \$3C480, which is %0000 0000 0000 0011 in the high word, so there were only 14 bits we could have shifted. As `V` was set, we skip to the label `pulTooBig` where we adjust `D0` from \$10 to \$08 and skip back to `pulNormalise` again.

On the second pass through the loop, we copy the last valid mantissa back into `D7`, and attempt to shift by 8 bits. This results in `D7` now holding the value \$3C48000 but this time, there was no overflow. As this was a successful multiplication of the mantissa, we subtract the correct powers of two from the exponent, which happens to be `D0`'s value, 8. Finally, we copy the new valid mantissa into `D6`, just in case, and drop into `pulTooBig` to divide `D0` down to 4, ready for another attempt.

On the third pass, we again copy the most recent valid mantissa back to `D7`, and multiply by 4 bits. This gives a result of \$3C480000 in `D7` and no still overflow, so we have a new valid mantissa. The exponent is adjusted by the correct powers of two and the new mantissa is copied to a safe place in `D6` before `D0` is again divided down to 2 at label `pulTooBig`.

The fourth pass through the loop multiplies `D7` up to \$F1200000 but causes overflow, we have to skip to `pulTooBig` to drop `D0` down to 1 for another attempt.

The fifth pass through the loop shifts `D7` to get a result for the mantissa of \$78900000 with no overflow. As before, the exponent is divided to cater for the multiplication, and the new mantissa is copied to `D6` for safety. At label `pulTooBig`, `D0` is divided down but is now zero. We are now holding a normalised mantissa in `D7`. The loop is finished and we drop to the label `pulNormalised`.

Remember when we calculated that only had 14 bits to shift to normalise `D7`, and shifting by 16 caused overflow? Add up all the successful shifts, $8+4+1$, then add one more for the initial doubling of `D7`, and you get 14.

Table 8.1 shows each pass and the various values as a summary.

[7] I have a funny feeling this code is something from Simon N. Goodwin!

Pass	D7 Input	D5 Input	Shifts	Overflow	D7 Output	D5 Output
Start	$0001 E240	$81F	-	-	-	-
Init	$0001 E240	$81F	1[8]	No	$0003 C480	$81E
1	$0003 C480	$81E	16	Yes	-	$81E
2	$0003 C480	$81E	8	No	$03C4 8000	$816
3	$03C4 8000	$816	4	No	$3C48 0000	$812
4	$3C48 0000	$812	2	Yes	-	$812
5	$3C48 0000	$812	1	No	$7890 0000	$811
End	$7890 0000	$811	0	-	-	-

Table 8.1: Floating point conversion of 123456

At this point, the exponent is $0811 and the mantissa is $78900000. Is this correct? We can convert the exponent to the correct powers of two by subtracting $800 to get $11, or 17 in decimal.

The mantissa, $78900000, is %0111 1000 1001 0000 0000 0000 0000 0000 but as we can safely ignore the trailing zeros, we have %0111 1000 1001 in the highest three nibbles. (Bits 31 - 20.)

The sign bit, bit 31, is a zero, so the mantissa is positive.

Bits 30, 29, 28, 27, 23 and 20 are set, but we need to subtract those from 31 to get bits 1,2,3,4,8 and 11 for out fractional powers of two. The mantissa is a fraction, in the format of 0.something, remember?

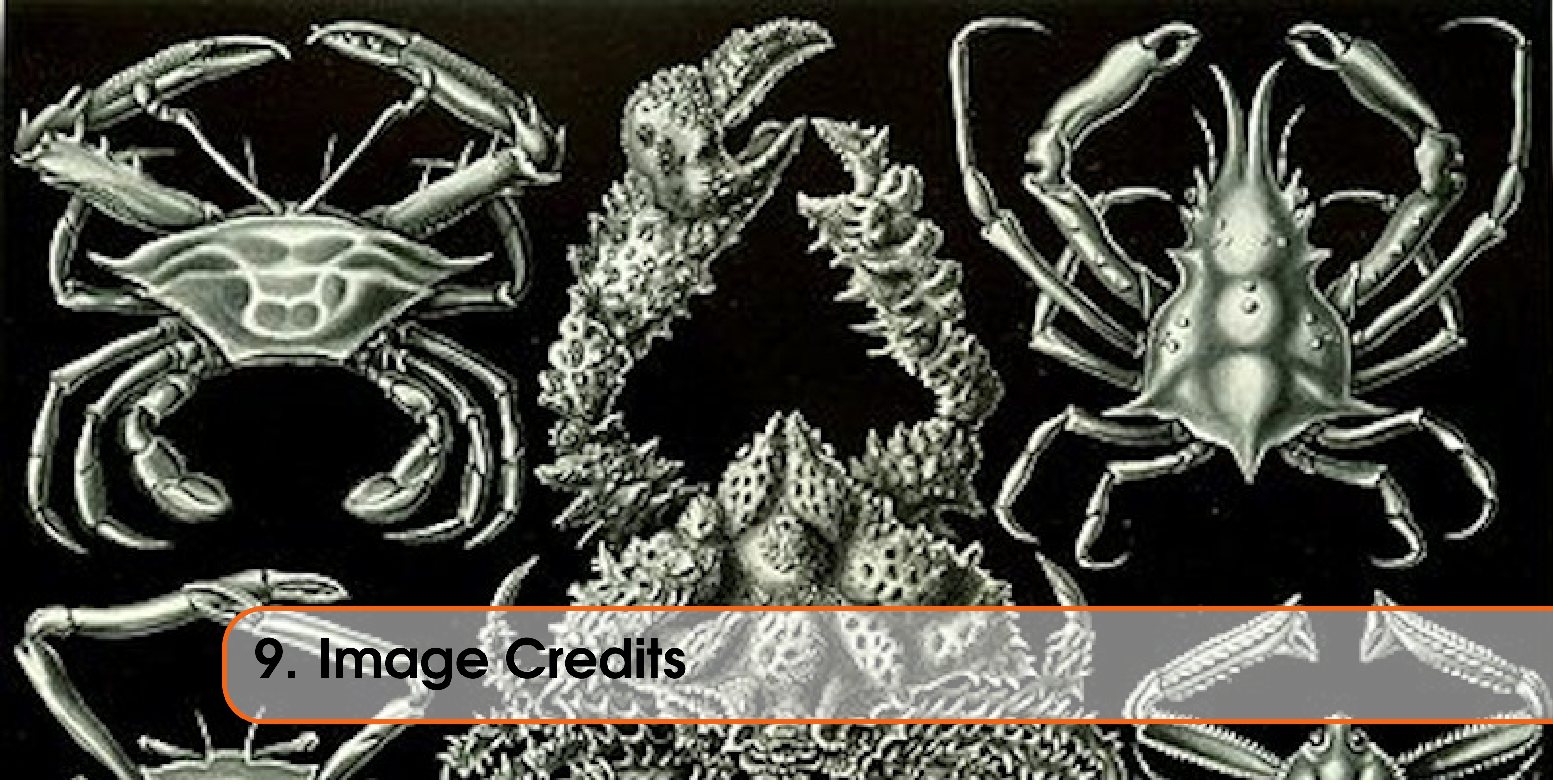
The bit values represent the powers of two that we take the reciprocal of and add them up, to get the actual fractional value for the mantissa. This sum is $\frac{1}{2^1} + \frac{1}{2^2} + \frac{1}{2^3} + \frac{1}{2^4} + \frac{1}{2^8} + \frac{1}{2^{11}}$ which works out as $\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \frac{1}{256} + \frac{1}{2048}$ and converting that to decimal fractions, not those nasty *vulgar*[9] ones, we get $0.5 + 0.25 + 0.125 + 0.0625 + 0.003,906,25 + 0.000,488,281$, giving a mantissa of $0.941,894,531$.

We can now work out the value as $0.941,894,531 * 2^{17}$which, funnily enough, is $123,456$, 2^{17} being $131,072$.

How easy was that then?

[8] It was an addition, but that's equivalent to a shift.

[9] Fractions in the format $\frac{a}{b}$in the UK at least, are called vulgar fractions.



9. Image Credits

The front cover image on this ePeriodical is taken from the book *Kunstformen der Natur* by German biologist Ernst Haeckel. The book was published between 1899 and 1904. The image used is of various *Decapods*. The Decapoda or decapods (literally "ten-footed") are an order of crustaceans within the class Malacostraca, including many familiar groups, such as crabs, lobsters, crayfish, shrimp and prawns. Most decapods are scavengers. The order is estimated to contain nearly 15,000 species in around 2,700 genera, with around 3,300 fossil species.

I have also cropped the cover image for use on each chapter heading page.

You can read about Decapods on Wikipedia and there is a brief overview of the above book, also on Wikipedia, which shows a number of other images taken from the book. (Some of which I considered before choosing the current one!)

Decapods have absolutely nothing to do with the QL or computing in general - in fact, I suspect many of them died out before electricity was invented, and the rest probably don't care about electricity or computers! However, I liked the image, and decided that it would make a good cover for the book and a decent enough chapter heading image too.

Not that I am suggesting, *in any way whatsoever*, that we QL fans are 10 legged crustaceans.